This electronic thesis or dissertation has been downloaded from the King's Research Portal at https://kclpure.kcl.ac.uk/portal/

Topologically Constrained Self-Organisation

Shaw, Matthew James

Awarding institution:
King's College London

The copyright of this thesis rests with the author and no quotation from it or information derived from it may be published without proper acknowledgement.

END USER LICENCE AGREEMENT

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International licence. https://creativecommons.org/licenses/by-nc-nd/4.0/

You are free to:

- Share: to copy, distribute and transmit the work

Under the following conditions:

- Attribution: You must attribute the work in the manner specified by the author (but not in any way that suggests that they endorse you or your use of the work).
- Non Commercial: You may not use this work for commercial purposes.
- No Derivative Works - You may not alter, transform, or build upon this work.

Any of these conditions can be waived if you receive permission from the author. Your fair dealings and other rights are in no way affected by the above.

Take down policy

If you believe that this document breaches copyright please contact librarypure@kcl.ac.uk providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 30. Jun. 2018

Topologically Constrained Self-Organisation

by

Matthew Shaw

A thesis submitted in partial fulfillment for the degree of Doctor of Philosophy

in the
Department of Informatics
School of Natural & Mathematical Sciences
King's College London

June 2014

Abstract

Autonomic computing suggests the need for dynamic systems that adapt their structure in response to environmental changes in a bottom-up fashion. Such systems can be considered to be composed of agents that self-organise in support of more effective operation. While various self-organisation approaches that aim to meet this need for continuous adaptation have been developed, these typically operate on structures that are not constrained to particular patterns (pipelines, hierarchies). Yet the ubiquitous use of such patterns when structuring task workflows, communication protocols, and traditional organisational design, suggests a need for their preservation when reorganising. In cases where specific patterns, or topologies, result from self-organisation, these are artefacts of the self-organisation mechanism, rather than the underlying topology itself being preserved. In contrast, this thesis explicitly tackles such adaptation, while accommodating the need to preserve topology.

The thesis introduces techniques for adapting a system's structure to improve task throughput, and builds on these techniques, to provide a means of preserving particular topologies. A framework for the reorganisation of defined topologies is introduced, and specific solutions are given for the case of pipelines and hierarchies, which reorganise to improve performance based on application-specific metrics, while preserving topology. Importantly, efficacy is only slightly diminished when topology is maintained, but at the cost of diminished autonomy.

Acknowledgements

First and foremost I want to express my sincere gratitude towards my first supervisor, Professor Michael Luck. Without his intense tutelage, unwavering guidance, and no small amount of patience, I would not be where I am today. Michael has boundless passion for all our work, which has inspired many in our college, and I know I speak for many when I say thank you for everything.

I also wish to thank my secondary supervisors Dr. Jeroen Keppens and Dr. Simon Miles, whose comments and discussions have helped guide and focus the development of my work. A special thank you to my close friend Samhar Mahmoud whom has always been there to discuss work, and throw around ideas. For always being there to listen when I need to vent, I want to thank Claudia Mazzoncini. Her words of encouragement and support are always appreciated.

Thanks to my family, my mother and father, Emma and Martin, my sister Catherine, and my brother Andrew and sister-in-law Natalie, who have supported me throughout my studies. In particular, a thank you to my sister, for reminding me on multiple occasions that she believes "Computer Science is not a real science!"

Thanks, also, to all lecturers, and colleagues in the Department of Informatics at King's College London, whom I have studied or worked with at some point or another. In particular, thank you to: Martin Chapman, Josef Bajada, Padmaja Sasidharan, Lina Barakat, Hai Xiao Liu, Vida Ghanaei, Ben Herd, Gareth Tyson, Emilia Maria, Peter McBurney, Sanjay Modgil, Elizabeth Black, Michal Sorka, Chiara Piacentini, Andra Voinitchi, Christos Hadjinikolis, Chris Haynes, Valeriia Haberland, Nathan Griffith.

Left till last, but certainly not the least, I would like to express a heartfelt thank you to Taffany Leung, for her support, love, encouragement, and unconditional confidence in me. She gives me the strength to believe I can achieve anything.

To you all, thank you.

To Taffany,

Contents

List of Figures

Chapter 1

Introduction

1.1 The Looming Software Crisis

The ability to communicate and interact is rapidly becoming standard functionality for all electronic devices. The capability to connect to the Internet — and thereby every other device in the world — was once solely for computers, but is quickly becoming commonplace on a multitude of household devices, from mobile phones to televisions and refrigerators. However, along with the increase in interconnected electronic devices comes the rather challenging task of managing them. At the personal level, keeping files, contacts, calendars and other pieces of data synchronised across telephones, laptops, and desktop computers at home, and in the office, can be a difficult task.

At an industrial level, the difficulty in information technology management is not just caused by the large number of physical devices (though maintaining these devices can be difficult in itself), but by the sheer number of virtual elements, such as pieces of software, that coexist in one massive virtual environment. Referring to this problem as the "looming software complexity crisis" [61], Kephart and Chess point out that the problem of managing all these pieces of software is not caused by any single piece, but by the complexity inherent in integrating a large number of heterogeneous software applications, such that the overall emergent behaviour is, at some times, difficult and, at others, impossible to anticipate.

Autonomic computing has generally been recognised as a tool to overcome this complexity problem [61]: it is a paradigm for large scale computing systems, such that entire

software and hardware infrastructures can be given high-level objectives and, meanwhile, autonomously complete low-level tasks such as configuring new devices, dealing with defective or unexpected software behaviour, optimising overall system operation, and protecting these large software ecosystems from any malicious entities, external or otherwise. Self-* (self-star) systems have been proposed to tackle each of these low-level tasks, where self-* refers to self-configuration, self-organisation, self-healing, or self-optimisation, to name a few [75]. In this thesis, we focus on developing the mechanisms required for distributed systems to automatically organise themselves, without external control or guidance.

1.2 Organisation in Telecommunications Networks

Consider a mobile telecommunications network, a highly distributed system that must cope with environmental changes, most prominently changes in the geographic distribution of mobile users, and changes in service demand. In these networks, mobile devices can be linked to any of a large number of access points, and when a mobile user moves from one geographical location to another, the mobile device can change the access point to which it is linked.

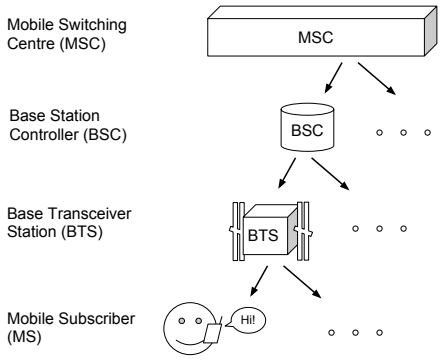


FIGURE 1.1: An example of a hierarchy in a mobile phone network

Beyond these access points lies a comprehensive network that is rigidly organised. As specified by the current standards specification for Universal Mobile Telecommunications Systems, there is a strictly hierarchical structure that must be maintained [36]. In this specification four types of device are used to construct a telecommunications network, known as *mobile subscribers*, *base transceiver stations*, *base station controllers*,

and *mobile switching centres*, as shown in Figure 1.1. The structure is hierarchical such that a *mobile subscriber* (for example, a mobile phone), is linked to one *base transceiver station* (or transceiver), but a single transceiver can be linked to multiple mobile phones. Likewise, a transceiver is linked to one *base station controller* (or controller), but a single controller is linked to multiple transceivers.

In more detail, a transceiver covers a geographic region known as a *cell*, and all mobile phones in that cell are linked to the network through that transceiver. In this context, a network consists of many transceivers, each of which covers a distinct *cell*, and all of these *cells* collectively cover larger regions so that a mobile phone can link with the transceiver that covers its current location.

Now, if a mobile phone changes location then it can break its link with one transceiver, and create a link with a new transceiver that covers the location to which it has moved. To oversee this change, groups of transceivers are managed by a controller, such that a single controller oversees multiple *cells*. When a mobile phone moves between *cells* that are both managed by a single controller, a process known as *handover* is performed by the controller that transfers the *mobile subscriber* from one transceiver to another. Finally, multiple controllers are overseen by a *mobile switching centre*, so that when a mobile phone moves between two *cells* that are not managed by a single controller, this switching centre handles the *handover* process. Moreover, this is the gateway to other telecommunications networks, such as those provided by other mobile phone providers, or the Internet.

While the organisational structure is rigid, the *handover process* provides a simple form of reorganisation at the lowest level of the hierarchy, which transparently transfers a mobile phone from one transceiver to another so that service is not intermittent. Essentially, the link between a mobile phone and a transceiver is automatically changed to ensure service continues. At every other level of the hierarchy, the structure is generally static.

However, if a controller is receiving many calls from all of its transceivers, it could be beneficial to change a link, so that one of the current controller's transceivers is managed by a different controller to distribute load. Alternatively, if two transceivers are overseen by two separate controllers then, whenever a mobile phone moves from the cell of one transceiver to the cell of the other, a *handover* process must be performed. Because each cell is managed by a different controller, the handover process must be overseen by

the mobile switching centre. If both cells were instead managed by a single controller, *handover* could be managed at the controller level.

Both examples suggest benefits that might be gained from a more flexible organisational structure that is able to adapt to particular circumstances. Yet given the scale involved (typically thousands of transceivers and controllers), central reconfiguration would be complex, requiring central monitoring of the entire network, increasing the costs of determining a better configuration of the organisation, and then requiring that the new structure be communicated. An alternative to a central solution, therefore, providing greater scalability might involve each entity in the network determining its own links to other devices in the network. The challenge in this possibility is that any changes must comply with the relevant system protocols. For example, the majority of existing telecommunications networks adhere to the Universal Mobile Telecommunications Systems Specification [36], which dictates that each mobile phone should be linked to a single transceiver. Similarly, each transceiver is linked to a single controller, and each controller is linked to a single mobile switching centre. This forms a strict hierarchical structure that must be maintained, but will be violated if it is not actively preserved. This is precisely the challenge we focus on in this thesis, in the context of more general systems.

1.3 Organising eScience

As an entirely different additional example of a distributed, and potentially autonomic system in which adaptation of the system leads to better performance, consider a large, potentially global network of electronic resources (such as devices, or even data), that are owned by different institutions, all of which are willing to pool their individual resources in order to gain access to a larger set of shared resources that would otherwise be unavailable to them. All resources are networked, with some performing computationally intensive tasks like the analysis of large scientific data sets such as those resulting from the Large Hadron Collider at CERN [52, 53]. Processing this data takes considerable time, and we want to process it as quickly as possible. If there is only one task, and it cannot be divided and processed concurrently across multiple machines, then the optimal solution is for the task to be processed on the fastest machine. However, if tasks are decomposable, finding a (near) optimal solution requires consideration of their distribution and added communication.

		Computational Devices					
		$sc1$	$sc2$	$dsu1$	$dsu2$	pa	lra
Services	Data Storage	N	N	Y	Y	N	N
	Processing Power	Y	Y	N	N	N	N
Hardware Specs	Memory	-	-	100	100	-	-
	Processing Power	Faster	Slower	-	-	-	-

TABLE 1.1: The capabilities and resources available in the eScience scenario

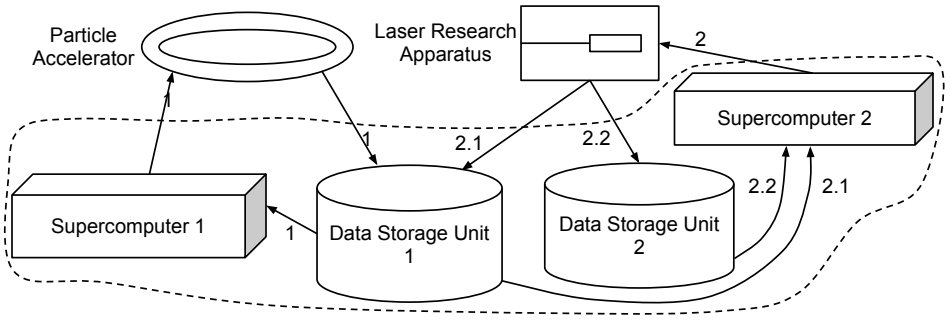


FIGURE 1.2: Visualisation of the eScience scenario

Now, suppose there are two research centres, each using a *particle accelerator* (pa), and a *laser research apparatus* (lra) respectively, and both generate large data sets. Such data needs to be stored and then processed, but neither research facility has the ability to do so. However, there are two data storage units, ($dsu1$ and $dsu2$), that are each capable of storing 100 units of data, and two supercomputers, ($sc1$ and $sc2$), that are capable of processing data, with $sc1$ processing data faster than $sc2$. This is summarised in Table 1.1.[1]

If pa and lra simultaneously perform experiments, respectively producing 50 units and 120 units of data, pa can store its data at $dsu1$, but lra stores 100 units at $dsu2$ and 20 units at $dsu1$, since neither dsu has the capacity for 120 units. pa's data is then passed to the fastest supercomputer, $sc1$, for processing and, since $sc1$ is now busy, lra's data is passed to $sc2$ despite it being slower. This is illustrated in Figure 1.2 where the arrows labelled 1 show pa's task being completed, and the arrows labelled 2 show lra's task being completed, where 2.1 and 2.2 show the task being decomposed.

While we will not discuss this further here, this more general example will be used later as a basis for considering how we may adapt organisational structures to particular circumstances.

[1]The capabilities of pa and lra are not relevant, so are omitted.

1.4 Dynamically Reorganising Distributed Systems

Generalising the telecommunications scenario above, distributed systems consist of multiple distinct entities that are linked to each other by some network. In the example above this is a hardware network, but in other cases this can also be a virtual network that overlays the hardware level. Through these links, distributed entities can communicate with each other. However, communication can become increasingly complex as these systems scale up. For example, a set of three or four entities can easily communicate with each other directly, but computational limitations, as well as limited bandwidth, can make direct communication between three to four thousand entities more difficult.

To reduce complexity, these entities need to be ordered or structured in some fashion. For example, by introducing tiers of authority like in a hierarchy, communication moving up the structure can be filtered by intermediate entities, to avoid the higher tiers being overloaded with information. Complexity can also be reduced by introducing entities that specialise in particular tasks. For example a *service registry* can be introduced with the sole purpose of keeping a record of all services in a system, so that it can later be queried for services. Such concepts are typically grouped under the banner of *organisation*, and involve increasing order in a system, such as structuring how these entities are linked to guide communication.

These organisational tools are typically employed at design time, such that the demands of the system are analysed *a priori*, so that the organisation of the system can be manually designed. However, the demands of the system may not be known beforehand and, even if they are, they may change. Distributed systems can be situated in dynamic environments, and the demands of such systems may change over time, so that the original design can become obsolete. When a dynamic environment changes slowly, it is plausible that when an organisation's design is no longer appropriate, it can be manually redesigned as required, but when the environment changes more regularly, such changes cannot be enacted manually, and so reorganisation must be performed automatically.

In particular, when the services that entities offer can change, or entities can come and go, the location of services cannot be known *a priori*, and so some form of service discovery must be used to locate them. However, services are located through communication, which is guided by the network that connects these entities. Therefore, to locate services we not only require some mechanism to do so, but also an appropriate organisational design to guide this search process. If entities change the services they offer, or if the

demands of particular services vary over time, then the organisation must be adapted automatically to reflect these changes.

A system's current organisational structure can be redesigned or adapted at run time, either in a top-down or bottom-up fashion. Any top-down approach can work towards a globally optimal organisation by enabling some central entity to gather information about all entities and how they are linked, and then command entities to change these links, accordingly. However, such centralised approaches are always limited by scale, and also operate with the assumption that some central entity can have overarching control of an organisation's structure, which is not always the case. Alternatively, bottom-up approaches can enable all entities to adapt their own links, so that the system-wide organisation is not explicitly designed, but instead emerges. Such systems employ self-organisation, and are referred to as self-organising systems; they have the benefit of distributing the strain of reorganisation across all entities in the system, so that reorganisation can be applied to a system at any scale.

1.5 Research Problem

To summarise, entities in large scale distributed systems are often linked by an organisational structure, but this structure can become obsolete as the environment changes. For such structures to be effective, links between entities must be changed at run time according to changes in the environment; for these changes to be effective at any scale, the structure cannot be controlled centrally but, instead, individual entities must control their own links, so that the overall structure of an organisation emerges from their individual actions.

However, in the case of systems with pre-determined (and required) organisational structure, such as the telecommunications network example above, where there are particular structural constraints (in this example in the form of a hierarchy), it is not clear that traditional approaches to self-organisation and emergence will be adequate. Instead, we need new mechanisms that will allow bottom-up reorganisation in an emergent fashion while at the same time preserving the required structural constraints. This is the focus of this thesis.

The aim of this thesis is thus to develop techniques for self-organisation of distributed systems while, maintaining particular structural constraints. More specifically, the thesis seeks to:

- provide an analysis of different organisation structures and their impact on service location;
- develop techniques for adapting structure to improve service location; and
- extend such techniques so that service location can be improved while at the same time preserving important (and in some cases fundamental) structural constraints.

By developing approaches to improve service location in distributed systems and, in particular, extending each technique to preserve structural constraints, we hope to enable systems, such as our telecommunications networks example, to make use of reorganisation approaches that would otherwise violate these constraints.

1.6 Publications

The work described in this thesis has already appeared in some publications, detailed below.

- M. Luck, L. Barakat, J. Keppens, S. Mahmoud, S. Miles, N. Oren, M. Shaw, and A. Taweel, Flexible Behaviour Regulation in Agent Based Systems, in C. Guttmann, F. Dignum, and M. Georgeff (eds.), *Collaborative Agents - Research and Development: Proceedings of the CARE@AI 2009 and CARE@IAT 2010 international workshops*, Vol. 6066 of *Lecture Notes in Artificial Intelligence*, 99–113, Springer, 2011.
- M. Shaw, J. Keppens, M. Luck and S. Miles, Towards a General Model for Adapting Structure while Maintaining Topology: Pipelines, in H. Aldewereld and J. Sichman (eds.), *Coordination, Organizations, Institutions, and Norms, Agent Systems VIII: Proceedings of the Fourteenth International Workshop, COIN 2012*, Vol. 7756 of *Lecture Notes in Artificial Intelligence*, 174–191, Springer, 2013.

1.7 Thesis Structure

The remainder of this thesis is organised as follows. Chapter 2 reviews relevant work in relation to organisation and related concepts, and discusses prior techniques for reorganisation at run-time. In Chapter 3, we motivate the development of a formal model of task allocation and execution with a particular scenario, and consider the process of collaboratively completing tasks within both the scenario and the model. The problem of locating services that need to be executed is identified as the key challenge, and this provides the basis for subsequent work. Chapter 4 introduces an evaluation environment, which is used throughout the rest of the thesis to analyse the performance of task completion. Then, using this evaluation environment, the next two chapters provide analyses of service location in centralised and decentralised contexts: Chapter 5 considers some common centralised approaches; and Chapter 6 considers a decentralised approach across several organisational structures, and in comparison to centralised service location. In seeking to address some of the limitations of static organisational structures, Chapter 7 introduces three approaches to reorganise at run-time, and evaluates the effect of each on a system's ability to locate services. Finally, since such reorganisation does not take account of underlying organisational constraints, Chapter 8 introduces a framework to adapt reorganisation so that it preserves topology. This framework is instantiated for two specific topologies: pipelines and hierarchies, and the performance of each is evaluated. The thesis concludes in Chapter 9.

Chapter 2

Organisation and Self-Organisation

2.1 Introduction

Self-organisation, in particular, has been recognised as a key aspect of autonomic computing, where self-organisation is a mechanism that allows a system to adapt its organisation at runtime without external control [104]. Though organisation is traditionally decided at design time, Carley and Gasser's review of computational organisation theory [23] suggests that there is no single correct or proper organisational design (Ishida et al. [59], Corkill and Lander [27], Lesser [74]) and, since then, others have agreed with this position (e.g., Horling and Lesser [56]). Importantly, they claim that being able to design an organisation is helpful, but adapting an organisation is essential [23].

This chapter separately reviews the fields of *organisation* and *self-organisation*, specifically in the context of *multiagent systems*, where an agent is an autonomous entity that is capable of flexible behaviour [124]. We begin by exploring the concept of *organisation* itself, by reviewing some common organisational paradigms in Section 2.2, and at least one application of each. In Section 2.3 we then distil these types into key properties, and discuss how varying degrees of these properties form the identity of each organisational paradigm.

This is followed by a review of the field of *self-organisation* in Section 2.4, covering a generally accepted definition, related concepts, and how *self-organisation* has been

implemented previously. Finally, we discuss *emergence* in Section 2.5, which, for many, is related to *self-organisation*, though the degree to which they are related is open to debate. The chapter concludes in Section 2.6.

2.2 Organisational Types

There is rarely a perfect organisational paradigm for any one situation [23]. For this reason there is often overlap between different types of organisations utilised at any one time [56, 57]. However, for clarity, in this section we review some commonly used organisational types (or organisational paradigms [56]) separately. The following sections each cover a single organisational type. Some key characteristics of each particular organisational type are highlighted, along with some applications of each. In particular, we review *hierarchies*, *holarchies*, *coalitions*, *teams*, *congregations*, and *societies*.

2.2.1 Hierarchies

One of the most intuitive forms of organisational structure is a *hierarchy* [56], which is the natural structure adopted by, for example, systems with authoritative (superior-subordinate) relationships [39, 82]. A hierarchical structure is a partial-ordering of nodes on some attribute that naturally decomposes into a rooted tree structure [82, 31]. Here, each node has an arbitrary number of child nodes connected to it via edges. For example, a small hierarchy is shown in Figure 2.1, in which the single node a_1 is the root of the hierarchy, with *subordinate* nodes a_2 and a_3 immediately below it. This structure has no cycles, and although in this example each node has two subordinates, a hierarchy does not necessarily limit the number of subordinates under one superior; an agent can have an arbitrary number of subordinates.

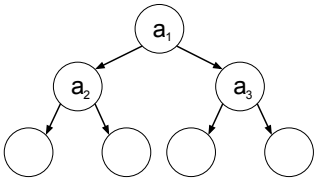


FIGURE 2.1: An example of a hierarchy

Because hierarchies are easy to interpret and are highly structured, they are regularly used in many contexts, such as in the structure of organisations [39], tasks [23], and agent plans [31].

Defining Characteristics When sets of agents are arranged in a hierarchy, the organisational structure can be representative of many kinds of relationship, but the most easily recognised is the *authority* of one agent over another. This is readily recognised because it can be seen at work, where managers have authority over employees or contractors, or in the military where commanders have authority over sub-commanders and regular troops. Following these two examples, node a_1 in Figure 2.1 is the *employer* or *commander*, while nodes a_2 and a_3 are *employees*, or *sub-commanders* (lower ranked officers), respectively. In this case, the direction of the edges reflects the order of importance or seniority in the hierarchy.

However, this ordering attribute could be something entirely different, such as the order in which services or capabilities are required [31], or lines of communication. In a hierarchy, communication lines can be bi-directional such that information is passed to superiors, and summarised as it moves up the hierarchy, while commands are sent down the hierarchy [14]. Nodes in the system can act as filters when this communication is taking place: as data is passed up the hierarchy each agent can filter the data to be propagated up the hierarchy [25, 40], ensuring that no node (especially the *root* node) is overwhelmed with information, thus avoiding *information saturation* [40].

As well as specifying *permitted* communication between agents, a hierarchy can also implicitly specify *restrictions* placed on system communication [56]. In Figure 2.1, node a_2 is not permitted to send messages directly to node a_3 and, likewise, node a_3 is not permitted to send messages directly to node a_2. Hierarchies, in their strictest form, are rigidly constrained, and the constraints are typically well defined, aiding in easy interpretation and implementation [80].

This can be seen in our military example, where lines of communication are rigidly upheld and constrained by one massive hierarchy with the Prime Minister or equivalent at the top, and soldiers of various levels below. In particular, the connections in the hierarchy govern strict lines of authority such that each soldier can pass information to the soldier directly above him, and can only give orders to soldiers directly below him in the hierarchy. Thus, if Figure 2.1 is part of a military hierarchy, soldier a_2 can pass

information to soldier a_1, and likewise agent a_2 does not expect to communicate with agent a_3.

This restriction is true in strict hierarchies, but Matheiu [79], and more recently Kota [70], relax this constraint, and introduce the use of cross hierarchical communication with peers to reduce the cost of communication up and down the hierarchy. This is equivalent to soldiers of the same rank in a group coordinating their efforts while performing small tasks that do not require supervision, but it may be expected that they keep their *superiors* updated on their current activities.

In fact, there are many variations of hierarchy. Fox describes three types: *simple hierarchies*, *uniform hierarchies*, and *multi-divisional hierarchies* [39]. A *simple hierarchy* is used when the number of agents in an organisation grows beyond the point where ad hoc organisation is no longer feasible, and so one agent is assigned the role of manager of all other agents, forming a two layer hierarchy. A *uniform hierarchy* creates multiple levels of management, where middle management acts as a filter of information and commands that are propagated up and down the hierarchy. Finally, when an organisation is too large to manage, even with the filtering effect of middle management, a *multi-divisional hierarchy* can be used to give each division (sub-hierarchy) autonomy for the moment-by-moment running of the division, while higher levels of the hierarchy plan towards long term, high level goals. Though hierarchies are used in many forms, they are in general regarded as rigid and fragile structures that are prone to single points of failure [80].

Applications of Hierarchies Because of their simplicity and predictable structure, hierarchies are commonly applied in many systems. For example, in the context of task allocation in problem solving agent organisations, Kota et al. [68] make use of hierarchies in two ways. First, they use a hierarchical *workflow*, which is a decomposition of a task into smaller sub-tasks, and each connection is a dependency between tasks, such that one task cannot start before the task above it is finished. Secondly, a hierarchy is used to organise agents in an organisational structure to guide service discovery. However, Kota et al. use a loose definition of a hierarchy, and allow peer connections that are utilised if the hierarchy is not sufficient to locate required services.

Kinnebrew and Biswas also decompose tasks into a hierarchical structure, to improve the efficiency of sensor webs [62], enabling some sub-tasks to be executed before others can be started. By interpreting Figure 2.1 as an illustration of such a workflow hierarchy,

task a_1 must be executed before task a_2, because a_1 is the parent node of a_2, and sub-task a_3 cannot be executed until a_1 is completed, for similar reasons. However, since a_2 and a_3 are siblings (this is to say that they are both children of the same parent), they may be executed either sequentially or in parallel. In these examples, the direction of the edges shows the order in which the tasks must be completed, so that the root node must be executed first, followed by its children, and so on. Here, *time of execution* is adopted as the ordering attribute, rather than authority.

In a slightly different style, Moffett makes use of a hierarchy to structure roles in a system [82], by which some roles are sub-roles of another. For example, if we interpret Figure 2.1 as a role hierarchy, a_1 could be a generic *health care provider*, while a_2 and a_3 could be a *nurse* and *physician*, respectively, where both are sub-roles of health care provider. Here, the hierarchy is used to classify roles semantically. Moffett's role hierarchy can thus be seen as using an attribute of each role to structure them [82].

2.2.2 Holons and Holarchies

First coined by Arthur Koestler [65], the term *holon* is derived from the two Greek words *holos* and *on*, which translate to *whole* and *part* respectively. The term refers to the duality of entities in almost all systems, whereby any entity can be simultaneously considered a whole in itself, and also a part of a larger system.

Defining Characteristics An example holonic structure can be seen in Figure 2.2, where each solid circle represents an agent, and each dotted circle represents a *holon*. Each *holon* is complete in itself, hence it is *whole*, yet it also forms a *part* of a larger system. Each holon is thus recognised as a distinct entity, rather than as its constituent parts. The structure of a holonic organisation is similar to that of a hierarchy, with the added property that each sub-tree generally has much more autonomy, and it is often the case that the sub-tree may appear to be a single entity to higher levels of the holonic structure [56]. In this respect, it is very similar to a *multi-divisional* hierarchy described by Fox [39], mentioned in Section 2.2.1.

Applications of Holarchies Holonic organisational structures have mainly been used in business and manufacturing domains, such that *holons* in a *holarchy* can be a group of workers in a team, a group of teams in a department, or a group of departments making

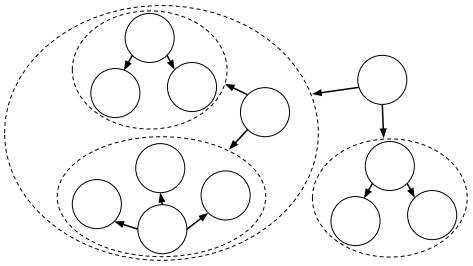


FIGURE 2.2: An example of holons in a holarchy

up a particular company. The factor that distinguishes a *holarchy* from a *hierarchy* is the partially autonomous behaviour of each *holon*, and the encapsulation of all lower entities into a distinct separate entity [56, 37].

2.2.3 Coalitions

A coalition is generally regarded as a set of agents that work together to achieve a specific, well defined goal, but each individual agent is only interested in improving its own utility [92]. They are usually short-lived groups that form to achieve a specific goal, and disband when the goal is achieved, the goal is deemed unachievable, or the coalition is simply no longer relevant. Coalitions are known for their loose organisational structure, making them extremely adaptable [3].

Defining Characteristics Three coalitions can be seen in Figure 2.3, where each solid circle is an agent, and each dotted circle surrounds the agents in each coalition. Each subset of the population of agents is a potential coalition [56, 94], and a coalition that contains the entire population of agents is known as the *grand coalition* [108], though this is generally regarded as inefficient as there are often large management costs in a coalition that increase significantly as the size of the coalition increases [24]. A coalition is usually formed around some mutual goal, and the agents in the coalition work collaboratively because it is either inefficient or impossible to achieve the goal separately [106, 72]. Though a coalition is formed around some mutual goal, each agent may have many goals of its own, and the coalition's lifetime is limited to the time it takes either to achieve the goal, decide it cannot be achieved, or until the goal is no longer relevant, at which point the coalition disbands [107].

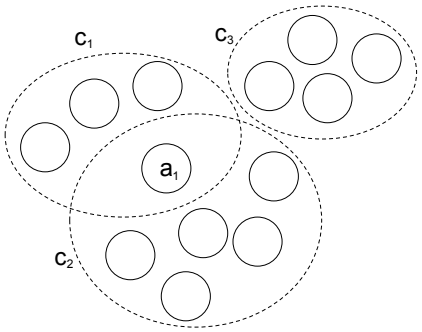


FIGURE 2.3: An example of coalitions

Some consider membership of a coalition to be mutually exclusive such that each coalition is distinct and disjoint [100, 120, 95]. However, Shehory and Kraus [108] argue that overlapping coalitions are possible, so that agents can be members of multiple coalitions simultaneously (shown by agent a_1 in Figure 2.3), though they point out that doing so adds complexity and possible inefficiencies, due to an agent's commitment being split. Intuitively, an agent cannot join two coalitions with conflicting goals. If an agent is a member of two coalitions, then it must consider how much time and effort must be allocated to each coalition of which it is a member.

Agents in a coalition are self-interested, but their mutual goal motivates cooperation for as long as the mutual goal exists. In this context, a division of some resulting utility must be calculated [101]. Griffiths and Luck relax the need for a mutual goal, arguing that an agent can join a coalition, even if doing so is not to the agent's immediate benefit, provided that agents are confident that there will be personal benefit in the long term [48]. Others also agree with this broader view, describing coalition formation simply as the coming together of autonomous agents to work together in a coherent group [97, 13]. However, as all agents are self interested, forming a coalition in this manner is all but impossible, because it can be unclear if these long term benefits will ever arrive. To combat this problem, Griffiths and Luck [48] employ a trust model so that when a coalition is being formed, agents can reason about the risk of working to their short term detriment, but long term benefit, based on past interactions with other agents.

Applications of Coalitions Soh and Li [115] introduce a model to formalise coalition formation, so that individuals that see the need for a coalition can plan for coalition

formation, enact the formation process, then evaluate the performance of the formation process and learn from it to improve later coalition plans. Johansson introduces the concept of norms to coalitions, where a norm represents an expected behaviour that has emerged over time [60].

Shehory and Kraus make use of coalitions when allocating tasks to distributed problem solver agents. They explore the use of both disjoint and overlapping coalitions, and present algorithms that can successfully form coalitions, and achieve almost optimal solutions when executing tasks in the blocks world domain [108].

Allsopp et al. explore the use of coalition formation techniques to aid in the logistics of international multilateral operations [3]. In scenarios where a number of nations, or other internationally recognised bodies (such as the UN, or Oxfam), each have short or long term interests in the outcome, producing some form of organisational structure is extremely difficult. Governments for each nation are, in general, self interested, and their interests do not always overlap. However, Allsopp shows that by applying approaches developed in the field of agent-based coalitions, effective and flexible command and control structures can be produced in this complex domain.

2.2.4 Teams

As a starting point when defining *teamwork*, Tambe appeals to a dictionary definition, describing it as "...a cooperative effort by the members of a team to achieve a common goal" [116]. However, given the review of coalitions in Section 2.2.3, this is too broad a definition to distinguish a *team* as a distinct organisational paradigm. Cohen et al. go as far as to say that a team of agents must have a shared goal and mental state, the result of which is unified activity, without which a team does not exist [26], though some believe these constraints to be too restrictive [7]. The main difficulty in distinguishing coalitions and teams is that their most distinguishing feature is not their actions, but the mental state that motivates these agents to cooperate.

Defining Characteristics Both coalitions and teams take part in achieving common goals. The distinguishing but subtle difference is that, in a coalition, agents work to achieve their own goals, and cooperate with other agents when these goals overlap. In contrast, agents in a team work together to ensure that the organisation as a whole achieves some system-wide goal [40, 116]. In some cases this means that an individual

agent will work to its own detriment if the organisation as a whole achieves its goal. Agents that are part of a team can also reason about the consequences of their joint actions, as well as the tasks that need to be completed to achieve a certain goal [116, 49].

As with all the organisations we review here, each team consists of a number of agents, and for each of these agents autonomy is arguably the most fundamental property. Yet agents in a team agree to collaborate, which almost always means a reduction in autonomy [16]. Therefore agents in a team are willing to give up their autonomy, at least to a degree, so that the team as a whole can achieve its goals [34], highlighting the trade-off in a team between the authority of a lead agent (if one exists), and the autonomy of team members.

Tambe agrees that the feature that sets a team apart from any other group of agents is *cooperation* towards a common goal, coupled with significant *communication* and *coordination* between agents so that cooperation can be achieved [116]. This high level of communication keeps all agents up to date about the activity of all other agents, thereby enabling close and efficient collaborative work.

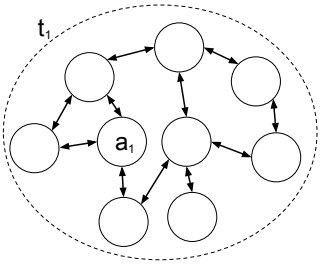


FIGURE 2.4: An example of a Team

The amount of communication needed to coordinate a team can be extremely heavy due to the intensive coordination required and, as a result, can be a drawback of such structures [90]. In addition, the pattern of interactions can be arbitrary, as can be seen in Figure 2.4, in which a high number of bi-directional connections between agents represents the large amount of communication links that are needed to coordinate a team [56]. Diggelen discusses the possibility that teams can vary across three dimensions depending on: the time leadership is established; whether planning is centralised or decentralised; and whether tasks are allocated to the team as a whole or individual agents [34]. However, some disagree with the notion that team planning can be centralised, instead arguing that team members tend to work together [40, 116], and reason about their actions as a group [49].

Applications of Teams Schurr et al. introduce a system called DEFACTO, to train *incident commanders* in the skills they need to oversee disaster response in time-critical scenarios such as natural disasters, or terrorist attacks [102, 103]. Their system makes the key assumption that each person in the rescue team communicates through an individual agent proxy, rather than communicating directly. As such, a potential incident commander does not need to know whether he, or she, is communicating with a real person, or an agent simulating a rescue team member. So, for training purposes, the DEFACTO system can make use of software agents to replace each person in a rescue team to simulate decisions the incident commander would need to take in a real disaster.

2.2.5 Congregations

Any academic can work in solitude, but in doing so their work will have no impact on the wider world, and their progress will be minimal. What makes academia productive is the collaborative work performed between multiple researchers, research groups, and institutions. However, finding people to collaborate with can be extremely difficult. For example, in 2011, employed academics made up approximately 0.3% of the population in the UK, and of that group, only 6.1% worked in engineering and technology subjects[1], so for a computer scientist to find a suitable academic to collaborate with by talking to random people in the population will take a long time. For this reason, most academics work in a university with other academics, and attend academic conferences to meet people with similar interests and capabilities. Such groups of people with similar capabilities and interests are known as *congregations*, and agent organisations can be grouped in the same manner.

Defining Characteristics A congregation is a group of entities that have characteristics or capabilities that are either similar or complementary in some way. For this reason, two distinct congregations typically have distinct characteristics, otherwise they would tend to merge into one congregation. Figure 2.5, inspired by Horling and Lesser [56], shows three congregations, where each solid circle is an agent, each dotted line represents a congregation, and the variation between each congregation is shown by the different shading of each group of agents.

Brooks and Durfee [19] describe a congregation as a group of self-interested agents, that congregate to decrease the time and effort needed to locate other agents with similar

[1]Statistics derived from data collected by www.hesa.ac.uk and www.ons.gov.uk.

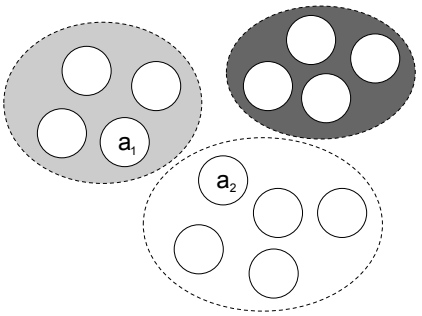


FIGURE 2.5: An example of three congregations

or complementary capabilities. Congregations are used in open environments: they are free to come and go. The success of a congregation does not rely on any individual agent, but rather the sum of all the agents that currently reside in it. Individual agents can therefore join and leave throughout the lifetime of the congregation, and as long as the congregation population size is maintained, the effect is minimal.

Agents join a congregation for some benefit, but the benefit received must outweigh the cost of being part of the congregation (cost of interaction and management, rather than any membership cost), or the agent will leave [17, 20]. For example, using our academic conference example, academics attend to meet other academics, learn about other people's research, and find possible opportunities for collaboration. However, to attend, funding is needed for flights and hotel rooms. The benefit of the collaboration must therefore be higher than these costs.

Though the agents that populate a congregation may vary over time, the capabilities of the agents themselves are typically stable [56]; for example a computer scientist will rarely decide to become a football player. Brooks and Durfee [18, 19] refer to congregations as long term entities, potentially surviving the agents that originally formed the congregation. In addition, agents that are part of the congregation for an extended amount of time repeatedly interact. Griffiths builds on Brooks and Durfee's concept of a congregation and, based on these repeated interactions, he enables agents to model the trust of others, which he views to be an agent's assessment of the risk of cooperating with another, based on previous interaction experience [46, 47]. This assessment is then used when forming a special type of *congregation* known as a *clan*.

Agents can be members of multiple congregations, as long as agents consider it to be beneficial. Brooks and Durfee [19] note that the decision to leave or join a congregation

is left to the individual agents, which they must do effectively as a bad choice of congregation affects their own performance far greater than that of others. They also consider those characteristics that are not dealt with in congregations. For example, there is no notion of payment for completing tasks, since congregations are not task-oriented.

Applications of Congregations Horling and Lesser describe and review *congregations* and *markets* as two distinct organisational paradigms [56]. However, according to Brooks and Durfee, a market is, at most, a specialised form of congregation, and is generally just a particular application of the congregation paradigm [18].

Brooks and Durfee use *congregations* to cut down the size of a marketplace, so that auctions can efficiently be established [18]. By cutting down the search space of possible buyers from the whole population to those that have congregated, they find that a system can achieve relatively constant profits, regardless of its scale.

2.2.6 Societies

Human societies are not typically concerned with how individuals are linked, what roles they plan, or what they plan to do at any given moment. Instead, societies consist of individuals that are bound to rules or norms. An individual in a society is expected to behave in a particular way, and also has expectations about the behaviour of others in the society. A society may thus expect individuals to drive on the correct side of the road, or join the end of a queue before paying for goods in a shop. Societies are also inherently open, with individuals of varying skills, knowledge, and cultures. In agent based societies, the same expectations exist, which help to decrease the complexity of interactions. However as societies are open systems, ensuring that agents comply with such expectations is difficult [109].

Defining Characteristics A society is generally understood to be a group of agents that have a common set of constraints, with which they are obliged to comply. Unlike all other organisational types discussed here, societies are inherently open [56].

In any system in which there is a large amount of interaction, such interactions can potentially be expensive. In systems with very specific and simple interaction protocols, communication is cheap, or at least communication costs are minimised. However, when

a system is open, interaction can be significantly more complicated, requiring negotiation over communication protocols, and the mapping of semantic concepts, causing communication to become computationally expensive. In such situations, centralised control is not feasible, because a central agent will present a significant bottleneck. Similarly, a completely decentralised system is not possible, because the number of negotiations required between agents for even simple interactions would be computationally prohibitive [109].

Motivated by this complexity, societies make use of *social laws*, *norms* or *conventions* to impose constraints on the population. These constraints guide agents, so that they know both how to act and what behaviour to expect from other agents in the environment. By limiting actions in this way, the complexity of reasoning about other agents is decreased [84, 109].

However, these constraints are only helpful if used effectively. Fitoussi and Tennenholtz [38] point out the importance of there being a balance between objectives and social laws: laws must be restrictive enough to have the desired effect — to constrain the system in the way they are expected to do — but they must also be flexible enough so that the objectives of all agents in the system are equally feasible. More specifically, the laws must restrict all entities fairly so as not give an unfair advantage to a subset of the population.

In addition, these restrictions are only useful if we can ensure that agents follow them. If laws are imposed, then an authority is required to enforce them in some manner [32]. If social norms are used, then all agents should encourage others to comply with any norms, or suffer punishment from society as a whole [77, 76].

Applications of Societies Willmott et al. present a permanent operating environment for agents known as *agentcities* ([123]). The goal of *agentcities* is to create an open system for even extremely heterogeneous agents with different goals and different levels of intelligence so that they have a common domain in which to act and communicate, despite their heterogeneous nature.

Using social laws can increase efficiency, but some means of ensuring that laws are adhered to is needed, or agents may simply disregard them and they become useless. One method of achieving this is through trust and reputation, as can be seen in the work of Ramchurn et al. [96], which uses mechanisms for agents to monitor the behaviour of

others according to previously agreed norms. If norms are followed then an agent is trusted, but if norms are violated then the offending agent is punished, either by direct sanctions (i.e. penalties such as fines) or by less direct means (such as competition from equally capable agents with better reputation).

2.3 Key Organisational Concepts

The term *organisation* is often used loosely to refer to any concept or paradigm that involves a group of entities. Indeed, Carley and Gasser comment that providing an all encompassing definition is difficult, citing the classic response to the question, "What is an organisation?" to be, "I know it when I see it." [23]. Even so, when the term *organisation* is used, a set of common concepts are often observed, and so in this section we highlight each.

According to Carley and Gasser [23] it is generally agreed that organisations, though applicable to most situations, are usually employed in large-scale systems. These are usually distributed systems with multiple entities that, to some degree, collaboratively work together to meet some goals. An organisation can have its own identity that is separate from any of the single agents it contains, and the use of organisation can enable the efficient use and coordination of resources, where resources can be anything from people to data or computational devices.

In the of context multiagent systems, Horling and Lesser give a more focused view, stating that "the organisation of a multiagent system is the collection of roles, relationships, and authority structures which govern its behaviour" [56]. In Section 2.2, we discussed generally accepted organisational paradigms, and in doing so touched on some more general organisational concepts. In this section we explicitly discuss the key concepts that are seen across each of these paradigms to varying degrees. We begin by discussing the relationship between goals at the organisation level and agent level.

2.3.1 Agent and Organisation Level Goals

It is generally accepted that agents are *autonomous*, goal-directed entities [124], and as such each agent has its own set of goals. At a higher level, organisations can also exhibit goals, though the nature of the goal is not always obvious.

The most general goal that drives most organisations is to aid in the completion of tasks, but each organisation tends to meet more specific goals when achieving this. *Teams* complete tasks efficiently through constant communication between team members. *Coalitions* can also complete tasks, but this is not the driving force behind a coalition. Instead, completing tasks is what drives the agents to act to improve their own utility, and the goal of the coalition is to enable self-interested agents to collaboratively execute tasks offering the incentive of increased personal utility. More indirect organisation level goals can be seen in *congregations*, which improve the efficiency of completing tasks by minimising the time and cost of service discovery. *Societies* decrease interaction costs by establishing norms to specify expected behaviour, and enforcing laws to prevent malicious behaviour.

2.3.2 Organisational Structure and Topology

In multiagent systems, agents are expected to be social [124]: that is, they can take part in complex communication such as coordination and negotiation, and they can build various relationships, or communication links with other agents if required. When organisational theory is discussed in the context of multiagent systems, the terms *topology* and *structure* are regularly used to refer to the overall set of links between agents [5, 119, 56, 127, 77]. However, these terms are at times used ambiguously, as we discuss here.

Argente et al. use *topology* as a collective term for the set of all links, and the term *organisational structure* is an all encompassing term, which encapsulates topology, roles, interaction models, and social norms [5]. Valetto et al. also use *topology* to refer to the links between agents in an organisation, and refer to *unstructured topologies* when stating that these connections are not constrained in any way [119].

According to Horling and Lesser [56], organisational structure guides lines of interaction, resource allocation, and authority, and when these organisations are constrained, such as in hierarchies, coalitions, and teams, it can aid the constituent agents by reducing the complexity of their reasoning, such as when deciding how to allocate tasks during the task allocation and execution process.

Zambonelli et al. [127] share this view, such that an organisation's *structure* is simply the set of all connections that exist between agents, and this structure can be constrained by some *topology* that specifies where agents must be situated in relation to one another.

Similarly, when studying norm emergence in networks of multiagent systems, Mahmoud et al. refer to *structure* as the connections that build up the network that connects agents, and they study how imposing *topology* on a *structure* affects the emergence of norms [77].

For clarity, we follow the interpretation of Horling and Lesser [56], Zambonelli et al. [127] and Mahmoud et al. [77], such that an *organisational structure* is a set of all links in an agent organisation, and a *topology* is imposed on an *organisational structure* to constrain or shape it in some manner. A hierarchy imposes the most tight constraints on an organisation's structure, such that all agents have explicit connections to a superior and a number of subordinates. Coalitions tend to be flat structures, with no explicit connections, though they sometimes have a lead agent forming an implicit hierarchy in the coalition. Connections in a team do not tend to be restricted in any way, but the agents in a team communicate regularly, and are often represented as highly connected graphs, where all agents communicating with all other agents is possible.

2.3.3 Task Environment

Though tasks do not seem immediately relevant when discussing organisations, the two are inextricably linked. Rosenschein and Zlotkin [98], and later Carley and Gasser [23], and Kota [66], assume tasks to be the key factor that defines the environment in which an organisation is situated, and they all refer to this as the *task environment*.

An organisation's *task environment* can vary greatly depending on the *volatility* and *repetition* of tasks, as well as the complexity of each. If the task environment is *volatile* such that the types of tasks are varied and often change, then organisational design is difficult because the needs of the organisation are changing. Conversely, if tasks are repetitive, then any organisation designed to complete these tasks is applicable for a longer period. Task environments tend to be: *oscillatory*, such as seasonal tasks; *incremental*, such as when new manufacturing technologies are introduced; or gradual, where only small changes are made [23].

Thompson [117] discusses how the overall group of tasks to be executed can vary over two dimensions, and the volatility of the *task environment* can be categorised as such. First, the task environment can either tend to be: *heterogeneous*, so that a large variety of services or capabilities are required; or *homogeneous*, so that only a few capabilities are required. Second, the task environment can be regarded as *stable* if the demand

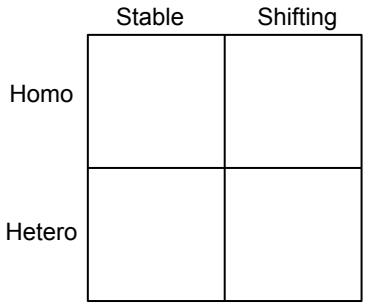


FIGURE 2.6: The range of possible task environments.

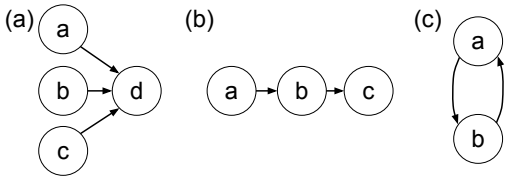


FIGURE 2.7: Three task dependencies: pooled, sequential, and reciprocal.

for services does not change, and *shifting* if service demand varies over time. The task environment can, therefore, vary over these two continua from homogeneous to heterogeneous, and from stable to shifting, and each varies independently from the other, as shown in Figure 2.6.

Tasks are usually decomposable into subtasks, which tend to have dependencies or temporal ordering constraints between them [117]. Thompson introduces three types of task dependencies, known as *pooled*, *sequential*, and *reciprocal*, each of which is shown in Figure 2.7. The *pooled* dependency can be seen in Figure 2.7(a), where task d depends on the completion of the set of tasks a, b, and c. Figure 2.7(b) shows two sequential dependencies, where task b depends on the completion of task a, and task c depends on the completion of task b. Finally, Figure 2.7(c) shows the reciprocal dependency between tasks a and b.

However it was later shown in the PCANS model that *pooled* and *reciprocal* dependencies are essentially *syntactic sugar*, and both can be decomposed to a set of *sequential* dependencies [71]. For example, in Figure 2.7(a) task d has three separate dependencies, a sequential dependency with task a, and task b, and task c. Figure 2.8 also shows how the reciprocal dependency between task a and task b can be decomposed into *sequential* dependencies by breaking each task down further so that each *sequential dependency* is

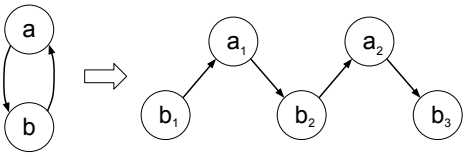


FIGURE 2.8: Decomposing a reciprocal dependency to sequential dependencies.

clear. Kota et al. make use of these sequential dependencies when modelling decomposable tasks, such that a task is only dependent on the completion of a single other subtask, creating a tree like structure of ordering constraints [66, 69, 70].

2.3.4 Roles

Hannoun et al. describe roles as a set of predefined or expected behaviours or capabilities that agents exhibit [51]. Similarly, Zambonelli describes a role as a well defined set of responsibilities or goals that are an aspect of the organisation as a whole [127].

Roles are seen in the organisational paradigms described previously. For example, in hierarchies all agents take the role of either superior or subordinate, and often both, depending on the context. For example, when an agent is communicating with its subordinates, it is fulfilling the role of a superior, but the same agent is, in turn, a subordinate when communicating with its superior. Though teams tend to be unconstrained with regard to expected roles, team members can elect a *team leader* that will be expected to take the lead in group coordination [116]. Brooks and Durfee [18] introduce a specialised congregation known as a market, which also has two roles: *buyer* and *seller*. As with hierarchies, these roles are not mutually exclusive; indeed it is common for agents to be both buyers and sellers simultaneously.

2.3.5 Overcoming Individual Agent Limitations

The key rationale for the use of organisational concepts is to overcome the limitations by which an individual agent is often bound [23]. Most well-known is bounded rationality: the notion that an individual agent is constrained either by its own capabilities, or its own knowledge. Carley and Gasser [23] divide agent limitations into four categories as follows.

Cognitive Limitations Cognitive limitations encompass any agent limitation due to incomplete information, or an inability to use the information available, effectively. For example, if an agent needs to trade on the stock market, it may have limited information, and it will not necessarily know what to do with the information it does have.

Physical Limitations Physical limitations for agents usually encompass: computational limitations, where an agent cannot process information fast enough; memory limitations, where it can only store limited data; or power limitations, if agents are mobile and have limited battery life.

Temporal Limitations Some high level goals take a long time to achieve, or are sometimes open-ended. In such circumstances, the effort to achieve the goal must survive past the lifetime of any single agent.

Institutional Limitations Individual agents can have legal or political limitations. If required resources are situated in multiple zones of authority, then any one agent cannot work alone. Instead, it must coordinate with agents that have appropriate authority to complete tasks.

Sandholm and Lesser consider bounded rationality when forming coalitions [101]. When deciding how to divide a population of agents into a number of coalitions, an optimal solution can be found, but since the complexity of finding a solution is *NP-complete*, these solutions are not tenable for any rationally bound agent. Allsopp et al. explore the use of organisational design for international coalitions in military scenarios [3]. Here, because entities from a number of countries are involved, boundaries of authority must be considered.

2.3.6 Organisation Life Cycle

Dang considers the use of coalitions in virtual organisations, and describes one possible interpretation of the life-cycle of a virtual organisation in which there are four distinct phases: creation, operation, maintenance, and dissolution [28], and all organisations go through these stages to some degree. Creation, otherwise known as formation, involves recognising the need for an organisation, and an organisation being established, and this process can be performed either at design time or at run time if the organisation can be dynamically created. Operation involves the use of the organisation, such as

collaboratively executing tasks, locating services, or other processes that can be aided through organisation. Maintenance involves adapting the organisation when it is clear that the current organisation is no longer fit for purpose. This can either involve the system being stopped, redesigned, and reimplemented, or the system can be adapted at run time using reorganisation or self-organisation techniques. This is discussed more in Section 2.4. Finally, dissolution, or disbanding is the tearing down of the organisation when it is no longer of use and adaptation is not an option or not relevant.

2.4 Self-Organisation

It is generally agreed that no single organisational structure is suitable for all situations, so an organisational structure is only suitable when it is designed in the context of a specific system. Where the environment is static, or fairly stable, an organisational structure can be specified at design time, but when the environment is more dynamic, any such structure can quickly become obsolete. To maintain its relevance, an organisational structure must therefore be equally dynamic. Therefore, the next logical progression in the organisation of agent-based systems is to move away from designing organisations, and instead design agents that can adapt their own links with others at run time, according to the needs of the system in which they are situated. These techniques are commonly grouped under the banner of *self-organisation* [104].

At its core, self-organisation is recognised as a mechanism that allows a system to maintain, and adapt its own organisation without external control [9, 55, 67, 104, 30]. Yet defining self-organisation is a contentious issue, as many disagree over what further characteristics are present in a *self-organised* system. Holzer et al. [55] believe that the two most important properties of self-organisation are *autonomy* and *emergence*, which is in stark contradiction to Wolf and Holvoet [30] who focus on the relationship between *emergence* and *self-organisation*, which they view as two related, but distinct concepts. When comparing these two concepts, Wolf and Holvoet note that in some cases the two concepts have been confused entirely, such that many properties are introduced under the banner of *self-organisation* when they instead describe *emergence*, such as in the case of Camazine et al. [22]. Serugendo et al. [104] take a different tack by defining self-organisation as:

> " ... the mechanism or the process enabling a system to change its organisation without explicit external command during its execution time."

They then divide the concept into two variations of *self-organisation* as follows. *Weak self-organisation* is a centralised mechanism to adapt a system's organisation that is internal to the system that it is reorganising. *Strong self-organisation* has no centralised control, external or otherwise, and hence organisation is an *emergent* property.

Self-organisation is a form of *adaptation*, oriented specifically towards adapting the organisational structure of a system of autonomous entities. This adaptation dynamically changes the structure according to its environment, or to changes in the goals of entities in the system. Such self-organising systems are, by nature, robust and so can continue operation in situations that would cause other systems to fail [67]. In what follows, we review the concept of *self-organisation*, by first discussing how *self-organisation* mechanisms can be categorised in Section 2.4.1, and then reviewing a number of existing techniques in Sections 2.4.2 to 2.4.5. Finally, some existing evaluations of self-organised systems are reviewed in Section 2.4.6. The concept of *emergence* is discussed in Section 2.5.

2.4.1 Categorising Self-Organisation Mechanisms

Many self-organisation approaches have been developed, and they can be broadly divided into approaches that are inspired by natural systems, and those that are not. In particular, Serugendo et al. [105] divides self-organisation approaches into five distinct classes, depending on the core mechanism that drives each, as follows.

Direct Interaction Self-organisation can be achieved through direct interactions between all elements being organised. This type of self-organisation is typically used in self-assembly and formation construction for mobile robots [78].

Stigmergy Self-organisation can also be achieved through indirect communication, known as *stigmergy*. Essentially, agents make changes to the environment, and other agents can observe these changes. This form of communication is common in natural examples of *emergence* and self-organisation, such as the use of pheromones when ants are foraging for food [88, 22].

Reinforcement Learning In scenarios where agents can dynamically change their behaviour, or capabilities, reinforcement learning techniques can be employed so that agents *learn* what capabilities are required depending on the requirement of the

environment around them, rather than through explicit coordination, or role allocation [105].

Cooperation Cooperative self-organisation mechanisms are based in closed environments, in which all agents are created by one entity. Such coordination permits agent behaviours that would otherwise be difficult to create. For example, the Organisation Self-Design framework [59] can merge two agents into one if their communication cost is too high, or divide a single agent into two if the demands of the system require it.

Generic Architecture The final type of self-organisation mechanism is such that the organisational structures that emerge are instances of generic reference architectures. These can subsequently be modified at run time to ensure that the organisation can meet the needs of the particular environment in which the organisation is situated [105].

In the remainder of this section, we review a number of existing *self-organisation* approaches. First, we discuss three approaches at length, each of which takes a different form. This is followed by a brief overview of some additional applications of *self-organisation*.

2.4.2 Past and Future Performance Heuristics

Kota et al. [68] provide of model of the *distributed task allocation problem*, which consists of a collection of agents that must work collaboratively to execute tasks that no agent can perform alone. Each pair of agents has one of four *types* of relationship, such that they are *strangers*, *acquaintances*, *peers*, or part of a *superior-subordinate* relationship. The different types of relationship determine what agents know about each other (the services that neighbours offer), and also determine the order in which agents are considered when one agent is locating another with which to collaborate. For example, an agent will consider allocating its task to its subordinates before considering its peers.

Importantly, task allocation can be expensive, and the cost is proportional to the number of agents considered in allocation process. So the set of links, or *relationships*, between agents, as well as the type of each *relationship*, can significantly affect the costs of the system. If an agent is a peer, then it is rarely considered during the task allocation

process, so the cost to maintain such a relationship is low. However, subordinates are considered first, so the cost of maintaining subordinate relationships is high.

To improve overall system performance (global utility), a *self-organisation* approach is introduced that adapts these relationships, thereby minimising the number of agents that must be considered when locating agents. This *self-organisation* approach is performed by all agents, such that an agent sequentially considers each of its relationships, and considers the benefit of every possible change that can be made to that *relationship*. The changes that an agent can make to a relationship are as follows: *remove peer relationship*, *form peer relationship*, *remove subordinate relationship*, *add subordinate relationship* or take *no action*. However, the changes are limited depending on the relationship; for example, agents that are already peers cannot form a new peer relationship.

To evaluate the benefit of making each of these changes, an agent uses a set of heuristic functions, which analyse the performance of the agent in the past based on its current relationships, and compares this to the potential performance of the agent in the future, if the change is enacted. An agent's performance heuristic is based on changes in the cost of communication, agent load, and the cost of changing a relationship. Once the benefit of each action has been assessed, the action with the greatest perceived utility is enacted. The utility of taking *no action* is always 0, so a change is only made if the utility of doing so is positive.

Since adapting relationships in this way incurs a cost, it is impossible to consider every relationship in the organisation at every time step. Instead, a dynamic method is used to adapt the *number* of relationships to be considered by an agent at each point. Thus, two issues are addressed: how to reorganise, and when.

This reorganisation approach adapts a system's organisational structure from within the system, yet agents only adapt their own relationships, so the overall structure is *emergent*. Additionally, this adaptation approach operates in a completely decentralised fashion. Therefore, according to the definition put forward by Serugendo et al. [104], this approach can be categorised as *strong* self-organisation.

2.4.3 Multiagent Reinforcement Learning and Self-Organisation

Learning can broadly be divided into three categories: *supervised*, *unsupervised*, and *reinforcement learning*. *Supervised learning* suffers from being over constrained, such

that the scope for learning is very small, and *unsupervised learning* suffers from being under constrained, such that the complexity of learning grows exponentially, and so is computationally prohibitive. As an intermediate approach, *reinforcement learning* can learn with no *a priori* knowledge, and the learning process is partially constrained by the learning algorithm itself, updating the direction of learning through trial and error at run time [113].

Though it can be argued that using learning techniques in the context of multiagent systems is simply an application of an already established field, some would disagree. Reinforcement learning was originally developed for environments that are Markovian, that is to say environments that can be modelled as a Markov Decision Process (MDP), but Nowé et al. state that when applying reinforcement learning in a multiagent environment, the MDP model is no longer appropriate [86]. Panait et al. also argue that multiagent reinforcement learning (MARL) should be studied separately for the following reasons. First, because of the number of entities in a multiagent system, the search space can become unusually large. Second, even small changes in the learning algorithm, or behaviour, can produce unpredictable effects at the macro-level [89]. In the context of multiagent systems, the goals of reinforcement learning can therefore change. Where reinforcement learning traditionally focuses on finding some globally optimal result, or behaviour, MARL must operate in environments that contain multiple learners. Other than in fully cooperative multiagent systems, MARL must work to achieve stability despite not having complete knowledge or control. In contrast to stability, agents must also explore to learn. However, rather than converge on a stable result, agents must instead learn continuously to adapt to the dynamic environment around them [21].

Abdallah and Lesser [2] make use of *reinforcement learning* techniques in the *distributed task allocation problem* domain. They acknowledge that though multiple multiagent reinforcement learning approaches have been developed to aid in effective task allocation, these typically assume that the organisational structure that limits agent interaction is static [1, 93]. Boyan and Littman [15] do consider dynamic systems, and introduce a Q-learning approach to network routing. However, they only explore the performance of their approach when network load is dynamic, keeping the structure of the network static.

Ziermann et al. [129] use reinforcement learning to reorganise communications paths in distributed embedded systems, with the aim of converging on a fair bandwidth allocation. However, their approach relies on initial exploration to find a suitable solution,

that is potentially not useful in the long term. This is because when self-organisation is used, the organisational structure changes, so an agent can be disconnected from a communication path, or agent, that it has learned much about, and can also be connected to a new path, or agent, that it knows nothing about, limiting the usefulness of its prior learning.

To tackle this problem, Abdallah and Lesser [2] develop an integrated approach consisting of a technique to adapt an organisation's structure, and a multiagent reinforcement learning technique to learn how to best allocate tasks. The self-organisation approach uses information from reinforcement learning to suggest which agent links should be changed, and the reinforcement learning technique uses heuristics to transfer what an agent has learned about agents with which it was previously linked to the agents with which it will be newly linked.

2.4.4 Distributed Sensor Networks

Distributed sensor networks (DSN) consist of collections of many inexpensive electronic devices. They are typically distributed across hard to reach areas, and are used to sense the environment, and wirelessly return what has been sensed or measured back to a base station or user. If the wireless range on a single device is limited, these devices can work together to ensure the network as a whole has connectivity to the base station. They can also work together to achieve some higher goal such as tracking, though achieving this can be difficult when these networks are unstructured and contain potentially thousands of devices [125]. The application of sensor networks can typically be divided into two categories, tracking [110], and monitoring [64].

Sims et al. make good use of self-organisation in a distributed sensor network (DSN) [111]. The DSN used is taken from a previously described problem [58], and consists of a set of cooperative sensor agents distributed across a geographical area. Here, each agent is in a fixed location, has local computational ability, and also has the ability to communicate with others, though bandwidth is very limited.

The total set of sensor agents is broken down into disjoint sets called *sectors*, each containing some sensor agents and one sector manager. The job of the sensor network is to detect moving vehicles that pass through an area, with the task of tracking one vehicle being given to a sector manager to accomplish. The sector manager must then use the sensor agents in its sector to complete the task.

All agents know their own location and orientation, but they can only sense the *distance* between the vehicle and their current position. This means that to track a vehicle, a minimum of three sensor agents need to be able to monitor it, in order to triangulate its position, requiring coordination between agents. For this, Sims et al. use a simple two-level hierarchy in which the sector manager is the authority over the sensor agents.

However, if the moving vehicle passes into another sector, the original sector manager is no longer be able to manage the task of monitoring it. At this point, coordination is needed between sectors, as there is no central authority. Instead, sector managers must negotiate with each other on even terms. To tackle this, a variation of the well known Contract Net Protocol [114] is used so that sector managers can self-organise to complete the tasks given to them in the most effective manner possible.

This method of adapting the system can potentially improve the efficiency of the network's tracking ability. If a sector is not able to track a vehicle it can try to pass the task to another sector that can do so. However, passing a task from one sector to another is costly, so the system attempts to delay passing a task to another sector as long as possible. It may be that the task must eventually be passed to another sector, but the cost of doing so is saved if the vehicle changes course and is still in range.

Coordination becomes even more difficult when multiple sectors need to work together, which occurs when a vehicle is on the border of the sector's range and a sector cannot track it alone. In this case, the task is not simply to pass control to another sector, but to enable the sectors to share agents to provide the minimum three that are needed to track a vehicle. This is done by negotiating with other sector managers and, if possible, taking an agent from another sector into the original sector so that the vehicle can be tracked.

Sims et al. also ensure that the system can cope with real-time dynamism, primarily the addition and removal of sensor agents, due to system failures or to more agents being added because of gradual network scaling, for example. Here, if a new agent becomes active, then it searches the area for a sector manager within its range. If one is found, then it joins the sector, but if not, then it becomes a sector manager itself, for the area around it. When more agents in the area become active, they then see the sector manager and join the sector. This is possible because agents are homogeneous and all have the ability to manage a sector if the need arises.

Agents can also cope with failures in other agents. All sensor agents periodically contact their sector managers, while sector managers periodically check in with all sensors under their control. If a section manager cannot contact one of its sensor agents within a time-out period, then the sector manager assumes it has failed and forgets it. If the sensor agent ever becomes active again then it must go through the normal start-up process, as described above. This means that even if a large number of agents fail, the ability to perform tasks may be diminished but the sensor network will continue to function.

If a sensor agent cannot contact its sector manager within a time-out period then it assumes that the sector manager has failed. Assuming that the system is asynchronous, one sensor agent in the sector will realise that the manager has failed before any others do, and will thus become a sector manager. This self-organisation creates an extremely robust system.

2.4.5 Other Self-Organisation Applications

Gershenson introduces reorganisation as a means of increasing the number of tasks a distributed system can execute in a given amount of time, by decreasing communication delay arising from both transmission (latency of sending messages) and work to be performed before a reply can be sent (decision delay) [43]. In essence, this is concerned with locating the agent a that suffers the most from transmission and decision delays combined, the agent b that is a neighbour of a and causes the most decision delays, and the agent c that causes the least decision delays in the whole system. Then, the link between a and b is removed, and a link is created between a and c. This technique was tested on four topologies (random-homogeneous, random-normally distributed, symmetrical, and scale-free), with results suggesting that: delay can be diminished, increasing the number of tasks executed; and the more links, the longer it takes to reorganise.

Gaston and desJardins introduce two techniques to adapt the organisational structure of a set of agents so that teams can more easily be formed to execute tasks [42]. This provides an initial attempt at adaptation based purely on organisational structure, rather than on application-specific information. The results show that structural adaptation can be effective, and leads to better performance.

2.4.6 Evaluating Self-Organisation

There have already been several efforts to evaluate structural adaptation (e.g. [70, 122, 128, 119]), each of which offers different insights into improving distributed task allocation to some degree, but the analysis of each approach is typically coarse grained, obscuring the distinction between different component parts of the task allocation and execution process.

For example, Abdallah and Lesser [1, 2] use reinforcement learning to adapt an organisation's structure, which successfully improves task throughput, but their evaluation only considers the overall reward value for the completed tasks, so that it is not clear how their approach affects the time it takes to locate services, nor its effect on agent load. Kota et al. [70] recognise that when completing tasks, multiple factors must be considered, such as communication cost, agent load, and the cost of reorganising. However, when it comes to evaluating their approach, these are consolidated into a single utility function, again obfuscating how each individual component of the overall cost is affected. Weerdt et al. [122] consider a similar problem, known as the *social task allocation problem*, and propose two methods: a centralised NP-hard solution, and a less complex decentralised version. Their evaluation is centred on a utility value for the quality of the overall global performance, and execution time for each algorithm, highlighting the benefit of the heuristic over its NP-hard counterpart. Zhang et al. [128] provide a strong but narrow analysis that makes use of a single evaluation metric (the average time needed to complete tasks), which is of little help in further analysis and improvement.

2.5 Emergence

In various fields (e.g. physics, sociology, computer science, biology, economics), the phenomenon known as *emergence* has been observed. Despite its appearance across a large variety of fields, in each the notion of emergence typically consists of a group of many interacting parts that cause some global property to emerge. This property is built from the culmination of all interactions, yet the interactions themselves show no design towards this emergent property.

In physics, a commonly studied example is Rayleigh–Bénard convection cells [44], where heating a liquid causes seemingly stochastic movements at the atomic level, yet globally,

structures emerge. Each atom rises when heated, drops when it cools, and bounces off other atoms in the process. These simple interactions cause multiple distinct structures to emerge, with no controlling force directing their construction.

Similarly, in economics, a stock market consists of a large number of self-interested traders that are motivated by their own improvement or gain. Yet what emerges is a robust mechanism for the allocation of effort that ensures a country produces all the resources it requires. This point is made succinctly by the economist, Adam Smith, when referring to such traders [112]:

> "He intends only his own gain, and he is in this, as in many other cases, led by an invisible hand to promote an end which was no part of his intention."

Though the definition of *emergence* in each field may vary, and each gives a varying degree of specificity, each typically recognises that such *emergent* systems [104, 10] exhibit some common properties and, in particular, some *emergent phenomenon* can be observed [30, 54, 22]. In this section a number of examples of *emergence* are discussed, drawn from some of the fields named above. Following this, we draw out some of the generally recognised concepts or properties of such systems, relating them to each of the discussed scenarios.

2.5.1 Conway's Game Of Life

Conway's Game of Life, initially introduced by John Conway, was later presented to a wider audience by Gardner [41] in his column, "Mathematical Games". The Game of Life, as discussed by Holland [54], has by no means useful functionality, but it serves as a good example of emergence. The game consists of a grid of cells, each cell being an individual location. Here, each cell in the grid is considered either *alive* or *dead*, and each cell alternates between these two states over time according to its own state and the state of its surrounding cells: its *neighbourhood*. This can be seen more clearly in Figure 2.9, in which the black cell is alive, the surrounding white cells are dead, and the surrounding white cells are the black cell's *neighbourhood*.

The states of all cells change throughout the game according to a common, simple behaviour that is employed by all cells in the system. For cells that are alive: each cell with less than two neighbours dies; each cell with four or more neighbours dies; and

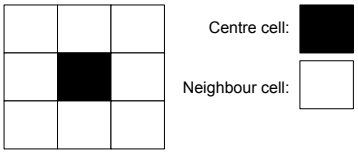


FIGURE 2.9: Example of a cell and its surrounding cells.

each cell with two or three neighbours survives. For cells that are dead: each cell with three neighbours becomes alive.

This is a zero-player game, so the initial configuration of live cells determines the result. Given certain initial configurations, collections or patterns may self-replicate. One example of this is the *glider* configuration shown in Figure 2.10, which replicates one cell down and one cell to the right each time the game is run through four times (i.e. every time all cells re-evaluate their state four times according to the above rules). This can easily be seen by comparing the first and last grids of Figure 2.10. Over an extended run of the game there appears to be, from a global viewpoint, one entity moving across the grid, and this apparent movement can be considered as an emergent behaviour.

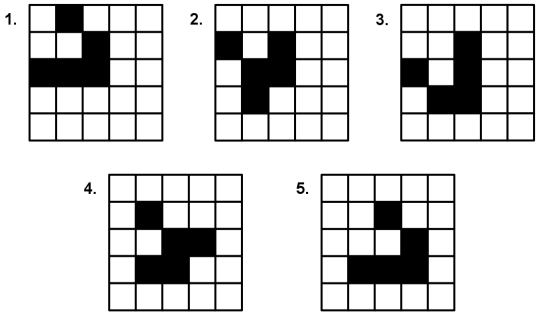


FIGURE 2.10: Game of life with 'glider' configuration.

Emergence can be seen in this example quite clearly, as a pattern emerges that was not an explicit part of the system's design. Each cell is a separate entity of the system, and the cells interact simply by being alive or dead in each other's neighbourhoods. From these simple interactions a global behaviour emerges in the system as a whole.

2.5.2 Simulating Emergence In Biological Systems

Emergence can be seen throughout nature, and the observation of these natural systems has inspired computational models, to aid in existing problems in areas such as networking infrastructure. For example, Panait and Luke show how a colony of ants foraging for food can find optimal paths without any centralised control or knowledge, and they imitate this mechanism in an agent-based simulation [88]. In their simulator, each agent is situated in a grid, and moves towards one of two possible locations: a food source, or the colony's nest. However, agents have no knowledge about the wider world, and they are unable to map a geographical space internally. Instead, agents must rely on *pheromones* that they leave behind themselves, and those that have been left behind by others, to guide them. Initially no *pheromones* exist, and so all agents randomly explore, but once an agent finds a food source it will begin to drop pheromones pointing to the food source as it returns to the colony's nest. As more agents find the food source, they increase the pheromone levels in its vicinity, gradually forming a *pheromone gradient*, so that in the future agents can move directly to the food source, rather than locate it through exploration. Additionally, because these *pheromones* are continually updated, if the food source is moved the *pheromone gradient* will, over time, change to direct agents to the new location.

The processes that the individual agents undertake (moving up pheromone gradients, updating pheromone levels according to the immediate environment, collecting food if it is available and dropping food if it is at the nest) are very simple, and the technique for interaction (dropping pheromones) is simpler still, but the actions that the ants take do not seem to have any direct relation to *seeking* out food. Instead, they move around according to the pheromones in their immediate vicinity, and when they come across food, they pick it up. Yet, what emerges is a completely new and coherent behaviour: the system as a whole seeks out a locally optimal path to and from food sources, and adapts this path if the location of the food source changes.

2.5.3 A Tale of Two Shops

Miller and Page [81] use a population of people that decide on a shopping location on a daily basis to highlight the difference between two scenarios that are similar, but only one of which exhibits emergent behaviour. In a town with a population of 100 people, there are two grocery shops, and each individual visits one of these two shops each day.

Given no information, each individual picks a random shop, so that each shop receives an average of 50 customers, the number on any day being subject to random variation. Here, a large number of entities are acting in parallel, but each makes its decision in isolation, so the even distribution of people across both shops, on average, is not an emergent property.

Now suppose that, by some oddity of nature, the town populace one day awakes realising that using one shop continuously is easier as they can learn the layout of the shop, but they are not fond of a shop that is overcrowded. Each day a single random person evaluates whether the current shop that they use is crowded, and if it is, they decide to switch. In this scenario, the presence of other shoppers is now affecting the decision to switch shops, changing the dynamics of the system, resulting in the number of people in each shop stabilising at 50 people. Each person has no interest in an even distribution of customers across both shops, and there is no central entity allocating people to each shop, yet an even distribution of customers emerges.

An additional property of this new shopping behaviour is that the system as a whole adapts to any perturbations that are introduced. For example, suppose that a customer in shop *one* starts to date a customer in shop *two*, and as a result decides to change shops so that they can shop together. This will break the stable 50/50 split between shops, but the next day a random shopper may feel overcrowded, change shop, and restore equilibrium.

2.5.4 Key Properties

Serugendo et al. review emergence in the context of self-organisation [105], inferring emergence through the properties that are present. Wolf and Holvoet also discuss, in detail, the distinctive properties that define emergence, and in particular what separates emergence from *self-organisation* [30, 29]. Therefore, our discussion here draws from both of these sources. In each of the following sections we discuss the importance of a number of key concepts to the overall notion of emergence, and tie each to the scenarios discussed in the previous section.

Micro-Level vs. Macro-Level Emergence can always be observed at two distinct levels, or from two distinct perspectives: the micro-level, which is concerned with the individual actions and interactions of the entities of the system; and the macro-level,

which considers the system at a broader, more abstract level, encompassing everything that can be observed globally in the system [105, 104, 118, 54, 91]. These levels are not necessarily explicit, and do not tend to be clearly divided in the system's design. Rather, they are different levels of abstraction from which the system can be observed.

In Conway's Game of Life [54], at the individual cell level only basic behaviour is observed, where each cell dies or returns to life depending on the number of cells that are alive or dead in its neighbourhood. Yet at the global level, patterns can arise such as the *glider* pattern that seems to move across the grid. Similarly, in the ant colonies example [88], each ant decides on a direction to move according to the pheromones in its immediate vicinity, and updates the pheromone levels in its current position according to pheromone levels in the surrounding area. Yet an optimal path is found to and from food sources.

Novel and Coherent Behaviour Any property or behaviour that arises is only an emergent phenomenon when it can be observed at the macro-level, so that the property or behaviour that emerges is novel with respect to the individual actions and interactions at the micro-level. If this global behaviour is produced through explicit action towards such a behaviour then it did not emerge, but was instead designed [81, 105, 104]. In addition, an *emergent phenomenon* must be observable for an amount of time. Often, when many entities interact, a number of novel patterns, behaviours, or properties may arise, some of which come and go. However, for a novel property to be recognised as an *emergent* phenomenon it must also exhibit stability, or longevity [87, 81] — it must be *coherent*. Wolf describes a *coherent* property as maintaining some *identity* over time [30].

Conway's Game of Life succinctly shows both novel and coherent behaviour [54]. In the *glider* example, it is impossible to predict, through observing each individual cell, that at the macro-level some global entity appears to move across the grid. In addition, the fact that this behaviour can be identified as a *glider*, shows that it is coherent. Conversely, many other initial configurations can be used that produce unstable behaviours, but these behaviours disappear from the system quickly, so are not coherent and are not considered to be *emergent phenomena*. In the shopping example [81], the distribution of customers is consistently maintained over time, and in the ant colony example [88], ants continually seek out a locally optimal path to food sources.

Robustness Emergent phenomena are typically robust to perturbation [81, 88, 30]. However, this does not mean that such behaviours are not susceptible to all perturbations. An example put forward by Camazine et al. [22] shows that wasps constructing a nest can produce malformed tunnels when they are disturbed in mid-construction.

In Conway's Game of Life [54], if two *gliders* move into the same space, then the *emergent phenomenon* is destroyed. However, in the two shops example [81], if a shopper moves from one shop to the other, then the system automatically restores equilibrium. In the ant colony example [88], obstacles can be placed on the current ant trail, and initially disrupt the system but, over time, a new path is formed.

Interaction Between Entities Miller and Page [81] discuss the difference between *disorganised* and *organised complexity*. When a large number of elements are taking actions in parallel but each is distinct, or each action is mutually exclusive, then this is *disorganised complexity*. In this case, according to the Law of Large Numbers, extremes are cancelled out, and so average behaviour over time can become extremely predictable, but individual cases are subject to random variation. Alternatively, *organised complexity* is when there is interaction between individual parts, changing the dynamics of the system, and often producing some *emergent phenomenon*. Wolf and Holvoet argue that without interaction it is impossible for emergent properties to arise. It is the interactions rather than the parallelism that causes emergence [30].

2.6 Discussion and Conclusions

According to Horling and Lesser [56], organisational structure guides lines of interaction, resource allocation, and authority, and when organisations are constrained, as in hierarchies, coalitions, and teams, this can aid the constituent agents by reducing the complexity of their reasoning, such as when deciding how to allocate tasks during the task allocation and execution process.

An organisational structure can be constrained to mimic the structure of tasks, and their respective workflows. Indeed, Thompson [117] believes the structure of a task is so closely tied to the structure of the agents completing the tasks, that he points out potential agent connections by highlighting the dependencies between tasks and subtasks, and Carley and Gasser explicitly state that an organisation is expected to

improve its performance if it matches the structure of the underlying tasks that it is executing [23].

The use case discussed by Zambonelli et al. [126, 127] also displays the connection between task structure and organisational structure in a manufacturing pipeline. Tasks are constrained such that each subtask is completed in sequence, with no concurrency. To reflect this, the organisational structure of the agents in the manufacturing process follow a pipeline topology. However, here the pipeline structure only holds for the flow of execution from one subtask being executed to the next. Zambonelli explains that a hierarchy can simultaneously be used to control the completion of tasks across the pipeline. Regardless of the structure used, both instances call for some topology to be used.

Krackhardt and Carley [71] recognise that the structure of any task can be broken down into a collection of ordering constraints, where the constraint (t_1, t_2) states that t_2 cannot be executed until t_1 is complete. Though these constraints can create many structures, they are often depicted as a tree structure. Given Thompson's assertions on the structure of tasks and organisations being interlinked, it is also easy to see how this tree structure can be transferred onto the structure of an organisation.

Finally, the Contract Net Protocol proposed by Smith [114] was developed to aid in the negotiation for task allocation in problem solving agents, where tasks are broken down, and each subtask is contracted out to other agents. This protocol again tends towards a task having a tree structure, but as the structure of the task is realised through contracts being successfully awarded to agents, it also implicitly creates a form of temporary organisational structure.

Constrained organisation is clearly prominent both in organisations and in concepts that influence organisational structure, such as tasks, workflows, and communication protocols. However, with the rise in interest of *self-organisation* and *emergence* and, in particular, the creation of biologically inspired self-organisation mechanisms, more rigidly structured organisations such as hierarchies are seeing competition from random or *emergent* structures, such as lattices and scale-free networks.

Moving away from constrained structures is neither an absolute positive nor a negative progression. If a system can operate and adapt effectively without considering structural constraints, this makes the design of self-organisation techniques much easier, because there are no constraints to limit adaptation. However, this is not always the case, as is

evident from our motivating example in Section 1.2. We can see that telecommunications networks must maintain a strict hierarchical structure if they are to comply with telecommunications network standards. If any existing self-organisation technique is applied, then the strict hierarchical structure will be broken, so current self-organisation techniques cannot be used in such systems. This is not to say that self-organisation in general cannot be of any use or benefit, but rather that in its current form, adapting some systems in this way is not tenable. For self-organisation to be acceptable in the context of the systems we are considering, we must design a method to adapt an organisational structure, while ensuring that any existing structural patterns, or constraints, are maintained.

Telecommunications networks, alone, provide examples of some of the largest distributed systems in the world yet, while they maintain particular structures, we nevertheless believe that the dynamic adaptation of such systems can be of great benefit. Therefore, in the remainder of this thesis we focus on how rigidly constrained, distributed systems can be adapted. Specifically, our ultimate goal is to develop an approach to enable a rigidly structured system to self-organise, while ensuring any structural constraints are maintained.

Chapter 3

Modelling Task Allocation and Execution

3.1 Introduction

By working together, agents can complete tasks that require a combination of services not offered by any single agent alone. Without such collaboration, tasks would otherwise fail. However, collaboration introduces complexity into any system, requiring tasks not just to be executed, but also to be managed in the sense that an agent must reason about whether it can execute a task and, if not, must allocate the task elsewhere. More specifically, when a task requires a set of services that no single agent offers, the agent that initially receives this task must determine which other agents can provide these required services.

Task completion thus requires a consideration of these different aspects, from the initial receipt of a task, through its direct execution if possible, the need to locate another agent to execute the task if not, its allocation elsewhere, and the communication that is required to facilitate all of this. If we are concerned with developing an effective and efficient mechanism to complete tasks in a distributed multiagent environment, we must therefore examine each of these aspects, in order to determine where the possibilities for improvements lie, and how they may be achieved. This chapter addresses just this problem, in elaborating task completion across its stages of task allocation and execution,

and in drawing out potential areas of interest that may provide opportunities for more efficient means of collaboration.

As we have seen in Chapter 2, it is clear that the connections between entities in large scale distributed systems are as important as the entities themselves. We therefore return to the eScience scenario presented in Section 1.3 and abstract out from it to develop a model in which we elaborate the key elements of the problem space, including both the entities and the links between them, and construct a formal model that represents the more general problem of agents that offer services, and tasks that require these services.

The chapter is structured as follows. In Section 3.2 the key elements of the eScience scenario, presented in Section 1.3, are extracted and formally described, creating a more general representation that we refer to as our *task allocation and execution model*. Then, Section 3.3 considers, step by step, the *completion* of a single task, and the path that a task typically follows in our eScience scenario as each individual part is executed. Section 3.4 describes the processes that are performed by each agent in our *task allocation and execution* model. The process of executing a task is divided into a number of distinct *phases* in Section 3.5, and then communication, a building block of each of these processes, is discussed in Section 3.6, as well as the cost of communicating between agents. Finally, in Section 3.7, we offer conclusions in relation to what can be done to improve the time needed to complete tasks.

3.2 Task Allocation and Execution Model

Though organisational adaptation may be applied in any number of contexts, for the sake of simplicity and understanding we focus on only one scenario in the domain of eScience. The eScience scenario described in Section 1.3 gives a single concrete instance of distributed entities, collaboratively completing tasks. In this section, key aspects are extracted from this scenario in order to develop a formal model that will be used as the base for subsequent work. We refer to this as our *task allocation and execution* model. This model includes similar characteristics to those introduced by Abdallah and Lesser [1], and Kota et al. [67], which are both concerned with competing tasks of some form through agent collaboration. Importantly, this provides some assurance that our model is not idiosyncratic but is instead one that applies well to a broad range of scenarios and ties directly to other work.

In the *eScience scenario*, different computational entities are required to undertake various tasks (such as $dsu1$ storing data for pa's task), and to pass these tasks to others if the entities themselves cannot execute them (such as $dsu1$ passing the remainder of pa's task to $sc1$, because it cannot process the data itself). The key *task* in this scenario is to analyse data. In this sense, to complete a task, some *requirements* must be satisfied, where a requirement amounts to a specification of a *service*, and the amount of work the service is required to perform. Tasks can be decomposed according to these requirements, potentially with ordering constraints: for example, to *analyse* data it must first be *stored* and then *processed*. Therefore, completing a task involves locating other entities that offer *instances* of these required services, in the some defined order.

In what follows the key elements from the problem space are formally specified. We begin by specifying the naming conventions used throughout this model. We then describe services, and how tasks are built up from the need for such services. Next, we discuss a simple mechanism to model the correlation between a set of requirements in a single task. Finally, we specify the structure of each device in our scenario, which we represent as agents.

3.2.1 Naming Convention

Throughout this formal specification, and the remainder of our work, some notation is used to formally represent various key elements. This notation is governed by the following conventions. Sets are always represented by one or more small, upper case letters, that form an acronym of the set's name, for example $\textsc{a}$ represents a set of agents. Elements in a set are represented by one or more italicised, lower case letters, such as an agent $a \in \textsc{a}$. There is one exception to the rule that elements are lower case: when we describe a set of sets, each element is represented by small upper case characters, in a similar fashion to all other sets. To help identify between multiple elements from the same set, a subscript is used to number each element. Finally, all functions are represented by strings with all characters italicised and lower case, except for the first character of the subsequent words making up the string, which are upper case (this is typically referred to as camel case). This is summarised in Table 3.1. For reference, this naming convention is repeated in Appendix A, along with a list of all terms used throughout the remainder of this thesis.

Notation	Description	Example
Set	Small, upper case letters. Acronym of set name	A
Set element	Italicised, lower case letters. Exception: set of sets	a
Multiple elements	Subscript is used to number elements	a_1, a_2
Functions	Full words, italicised characters. Uses camel case.	$addItem()$

TABLE 3.1: Naming conventions

3.2.2 Task Model

In our example, the system as a whole contains a set of *services*, each of which defines a piece of functionality, such as such as storing or processing data. The details of such services are unimportant for our purposes, and we simply specify the set of all services, $\text{S} = \{s_1, s_2, \dots\}$. Entities $dsu1$ and $sc1$ offer *instances* of these *services*, where $dsu1$'s data storage *instance* stores data from pa's task, and $sc1$'s data processing *instance* processes the same data. Though we can envisage a service instance to be specified by many parameters, we are only interested in the service of which it is an instance, and so a service instance $si = (s)$. Clearly, in order to fulfil tasks, these *service instances* must perform some work. A *requirement* is a specification of a *service* and the amount of work needed from an instance of the service in order to satisfy the requirement. An example of this is $dsu1$ storing 50 units of data, and $sc1$ processing 50 units of data.

For simplicity, we assume that all service instances provide the same amount of work, or effort, per unit of time, and we use time as a simple proxy for an amount of work. In this way, a service may be required for 3 units of time, while another requirement may specify that the same service is required for 6 units of time. A requirement r thus takes the form (s, srt), where $s \in \text{S}$ is the service of which an instance is required, and $srt \in \mathbb{Z}+$ is the amount of time for which it is required. Enactment of a service instance to satisfy a task's requirement is encapsulated as a *running task* $rt = (t, si)$ where si is the service instance executing task t.

Tasks are decomposable according to their requirements, as indicated above, and there are ordering constraints between these requirements that form a tree structure. We call the set of all ordering constraints in a single task a *requirement ordering tree*, ROT, an example of which can be seen in Figure 3.1, where each vertex, r_1 to r_5, indicates a requirement, and each edge indicates an ordering constraint between requirements. Each constraint takes the form (r_x, r_y), such that $r_x, r_y \in \text{R}$, and r_y can be satisfied immediately after r_x has been satisfied. In Figure 3.1, there exist two constraints

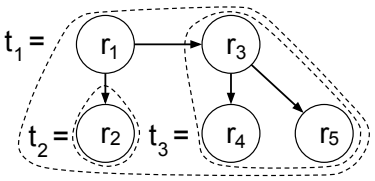


FIGURE 3.1: Example of a task, and a requirement ordering tree.

$rot_1, rot_2 \in$ ROT, such that $rot_1 = (r_1, r_2)$ and $rot_2 = (r_1, r_3)$, which specify that r_1 must be satisfied in full before r_2 and r_3 can begin (because r_2 and r_2 are children of r_1), but once r_1 has been satisfied, requirements r_2 and r_3 can be satisfied concurrently (because they are siblings). The full *requirement ordering tree* can thus be represented as ROT $= \{(r_1, r_3), (r_1, r_2), (r_3, r_4), (r_3, r_5)\}$.

A task t is thus a tuple, (R, ROT), where R is the set of requirements for completing task t, and ROT is the *requirement ordering tree*, that links these requirements. In each task there is only one requirement that does not depend on another, at the root of the requirement ordering tree. This is r_1 in Figure 3.1, and is known as the *initial requirement*. On this basis, a task can be divided into subtasks, where a subtask consists of a subset of these requirements, with relevant ordering constraints. A subtask is simply a subtree of the requirement ordering tree, such as tasks t_2 and t_3 in Figure 3.1, where $t_2 = (\{r_2\}, \emptyset)$ and $t_3 = (\{r_3, r_4, r_5\}, \{(r_3, r_4), (r_3, r_5)\})$. Finally, a task that has one requirement, and thus no dependencies, or an empty *requirement ordering tree*, we call an *atomic task*, of which task t_2 is an example.

3.2.3 Requirement Correlation

It should be clear from the above that the main structure in a task is a *requirement ordering tree*. The services that are required in this ordering tree could be arbitrary but, fortunately, this is not typically the case. Indeed it has been suggested by Guo et al. [50] that if multiple services or capabilities are required for a single task, the first service required tends to suggest what other services or capabilities will also be required. Therefore, one requirement can give an indication about the other services that are required in the same *requirement ordering tree*. We refer to this phenomenon as *requirement correlation*.

For example, when a Londoner organises a holiday to Hong Kong, a flight must be booked. Following this, there is a strong possibility that a taxi, coach, or train to

the London airport will also be required. Similarly, a hotel room in Hong Kong may also be needed, as well as some mode of transport on arrival. Conversely, it is extremely unlikely that a service to store or process large amounts of data is required when booking a holiday.

The same phenomenon is present in our eScience scenario. Both devices *pa* and *lra* have tasks that require data to be analysed, and this analysis involves processing large amounts of data. It is easy to see that if large amounts of data need to be processed, then they also need to be stored. We can also assume that a service will not be needed to book a flight from London to Hong Kong. Similarly, in the context of web services and service composition, Guo et al. have recognised the benefit of considering the correlation between multiple services. For example, when an institution owns multiple service instances, it may prefer to use its own services rather than those offered by others. Moreover, relationships between institutions can facilitate more efficient composition of multiple services, without affecting quality, encouraging agents to use a subset of services [50].

This observation is crucial to our work, since we will seek to take advantage of this phenomenon in addressing issues in the efficient and effective management and processing of tasks. For now, we simply assume some regularity or pattern in the tasks we consider, mirroring those in the real world, and observed by others in the context of service composition. Given this, and in order to proceed, we seek to model this assumed correlation between required services by insisting that a single task only ever requires services from a subset of all services, known as a *service category*, SC. For example, suppose that agents offer instances of services from the set of services $\mathrm{S} = \{s_1, s_2, \ldots, s_{20}\}$, and this set of services is divided into four distinct and non-overlapping service categories, as shown below. In this way, a task that requires multiple services can require a service from SC_1 or SC_2, but not both.

$$\mathrm{SC}_1 = \{s_1, \ldots, s_5\}$$
$$\mathrm{SC}_2 = \{s_6, \ldots, s_{10}\}$$
$$\mathrm{SC}_3 = \{s_{11}, \ldots, s_{15}\}$$
$$\mathrm{SC}_4 = \{s_{16}, \ldots, s_{20}\}$$

3.2.4 Agent Model

Tasks themselves are executed by entities providing services; for example in our *eScience* scenario, tasks are executed by computational devices. Each of these entities that executes tasks is encapsulated as an *agent*, which uses its service instances SI to execute tasks in its list (ordered set) of tasks, T. Each agent has a set of service instances $\text{SI} = \{s_1, s_2, \dots\}$. An agent can satisfy a task's requirement if it offers an instance of the service specified by the requirement. Once an agent begins to satisfy a task's requirement, the task becomes a *running task*, and remains so until the requirement is satisfied. To represent this, a *running task*, $rt = (t, si)$, is added to the agent's set of running tasks, RT, where t is the task being executed. An agent's running task capacity, $rtc \in \mathbb{Z}+$, limits the number of tasks it can execute at one time, so $|\text{RT}| \leq rtc$. This represents a limit of resources such as memory in a data storage unit. Once all of a task's requirements are satisfied, it is removed from RT, allowing more tasks to be executed.

This covers the execution of tasks, but since we are concerned with the allocation of tasks as well as their execution, we need also to consider the structure in which agents are connected to each other. In any system, services are provided and consumed via the links or *connections* between them. Moreover, for agents in a multiagent system to complete tasks collaboratively with others, they must interact with these others. Thus, an agent has a set C of *connections* to other agents. These *connections* are symmetric, so that if an agent a_1 has a connection to agent a_2, then agent a_2 also has a connection to a_1. Such connections are the means by which agents interact to provide or consume services and, more generally, to collaborate. A connection between an agent a_1 and a_2 is represented as $c = (a_1, a_2)$ such that an agent's set of all connections is represented as $\text{C} = \{(a_1, a_2), (a_1, a_3), (a_1, a_4), \dots\}$, and $(a_1, a_2) = (a_2, a_1)$.

Taken together, we define an agent a as a tuple, $a = (\text{SI}, \text{T}, \text{RT}, rtc, \text{C})$, where SI is a set of service instances that a offers, T is the set of tasks that a has received, RT is the set of tasks agent a is currently executing, rtc is the maximum number tasks that agent a can run at any one time, and C is a set of connections which agent a has between itself and others. Since any system of the kind we are considering has many agents with different connections, the system as a whole is considered to be a multiagent system, mas, such that $mas = (\text{A}, \text{C})$ where A is the set of all agents, and C is the set of all connections between agents.

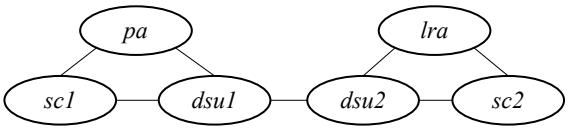


FIGURE 3.2: The initial structure in the eScience scenario.

3.3 Task Completion in the eScience Scenario

As we have indicated, the *completion* of a task in our model requires an agent to receive the task, determine if it can be executed, and allocate it to other agents if it cannot until, eventually, the task and all of its subtasks are executed and therefore complete. By returning to our eScience scenario of Section 1.3, we can illustrate this with the collection of agents *pa*, *lra*, *sc1*, *sc2*, *dsu1*, and *dsu2*. Suppose these agents, shown in Figure 3.2, are currently running and have already completed several tasks they have received.

Now, suppose that all these agents are idle when *pa* receives a task t that requires data to be *stored*, and then *processed*. Initially, *pa* considers t's *initial requirement*, which is to *store* data, and checks if it has the capability to meet the requirement. Because *pa* offers no data storage service, it must seek to *locate* another agent that does offer data storage, and does so by undertaking a search of the available agents. Such a search may be achieved in different ways, for example by querying a centralised service registry, or by querying individual known agents directly, asking whether they offer the required service. If, eventually, *pa* locates agent *dsu1* that offers the required service, then *pa* *allocates* the task to *dsu1*.

When *dsu1* receives the task, the task must be *executed*, but execution is not possible until *dsu1* has capacity to do so. If *dsu1* is already executing a large number of tasks when t arrives, then it must complete some of these tasks before it can begin to execute t. However, as we have indicated, *dsu1* is idle, and does have capacity, so begins *executing* t by satisfying its first requirement: *storing* the data. Finally, when this requirement is satisfied the next requirement must be satisfied (*processing* data), and so *dsu1* performs the same process that *pa* did previously. First, *dsu1* checks to see if it has the capability to process the data, and comes to the conclusion it cannot, and so proceeds to locate an agent that can do so. When searching, *dsu1* finds that *sc1* offers the required service, and therefore *allocates* the task to *sc1*.

As with *dsu1*, once *sc1* has received task t, *sc1* waits until it has capacity to execute t, then meets the next requirement in the task: *processing* the data. Once *sc1* has satisfied the requirement, *sc1* finds that there are no more requirements to be executed, and returns the results of the completed task back to *dsu1*. In turn, *dsu1* returns results to *pa*. At this point *pa* can be satisfied that the task is *complete*.

3.4 Allocating and Executing Tasks

Moving beyond the structure of each of the elements of our model as presented in Chapter 3, we are able now to consider, in relation to our model, the behaviour of agents so that tasks are completed, either by a single agent executing them in full, or through multiple agents executing distinct parts of a task. Here, agents are considered to be heterogeneous, with each agent offering a different set of services. However, the core processes that each agent performs are the same. All agents continually perform two processes: they analyse received tasks to determine whether they can be executed, and if possible execute them; and they reallocate any tasks that cannot be executed, to other agents. We consider each of these separately.

As stated earlier, in considering the management and execution of tasks, we adopt a base model similar to that of others (e.g. [70, 2]). For simplicity, we assume that time is discrete, with a number of time steps, in each of which an agent a must: manage its tasks list T; and execute tasks in its running tasks set RT. In managing tasks, a first places any received tasks in the list T. Then each task $t \in$ T is reviewed: if t's *initial requirement* can be satisfied directly by a (because a offers the required service and has the capacity to do so), a removes t from T, and adds the tuple (t, si) to RT where si is the service instance executing t; if a can satisfy t's *initial requirement*, but does not have capacity to do so, then the task remains on the list, waiting for capacity to become available; finally, if a cannot satisfy t's *initial requirement*, then it must *allocate* the task to another agent.

After T is updated, execution begins on an agent's running tasks list, RT, by satisfying each task's *initial requirement*. Since tasks require a service for a specified time (in time steps), *srt*, they remain in RT until *srt* has elapsed, at which point they are removed. At this point, if the task's *initial requirement* r_x is also the task's only requirement, then the task is complete, otherwise a subtask is created for every branch from r_x in the *requirement ordering tree*. The initial requirement of each subtask is $r_1, r_2, \ldots, r_n$,

such that there are n subtasks, and $(r_x, r_1), (r_x, r_2), \ldots, (r_x, r_n) \in \text{ROT}$. Each subtask is added to T so that it can subsequently be executed or allocated. Through the repeated use of this process, each task an agent receives will be managed, and either executed or reallocated to another agent.

Finally, when an agent cannot satisfy a task's requirement itself, the task must be allocated to another agent. For an agent to allocate a task it must, first, locate another agent that can execute the task and, second, reach an agreement with the other agent, to ensure the remainder of the task is completed.

Locating an agent and forming an agreement can be a complex problem, because we consider the agents in our system to be autonomous. Obtaining information about the services that other agents offer, and reaching an agreement about executing tasks, can potentially introduce the need for some form of negotiation. Though the issue of negotiation between autonomous entities is an important one, we instead assume agents to be, to a limited degree, cooperative. Therefore, agents will help each other to acquire information about services offered by themselves, and others, if required. All agents also cooperate to ensure that tasks are completed, and so when an agent a_1 wishes to allocate a task to a_2, then a_2 will accept.

Given that agents will cooperate in the allocation of a task, the key aspect is to determine to which other agent the task will be allocated. More specifically, if an agent does not offer a task's required service then it cannot execute it, so it must locate another agent that does offer the required service. We refer to this as the *service location* problem. Service location is an integral part of collaborative task completion. If agents are unable to locate services, then any task that requires a set of services that no single agent provides cannot be completed. In the chapters that follow, a variety of service location approaches are introduced.

3.5 Phases of Task Completion

By describing the completion of a single task in Section 3.3 we can see that completing a task is not just a matter of executing it. Importantly, *task completion* is built up of a number of processes, each of which makes up a portion of the overall time to complete a task. This section divides *task completion* into a five parts, or phases. Each of these phases is introduced and discussed separately below. The total time to complete a

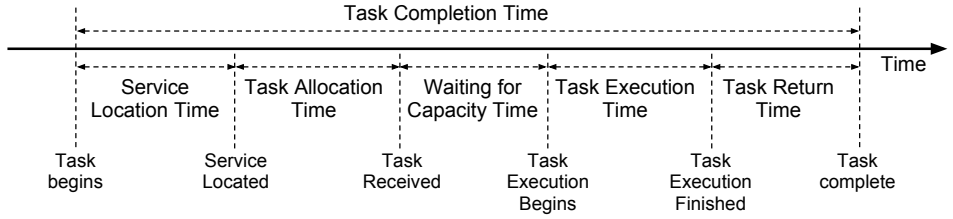


FIGURE 3.3: A summary of task completion time.

task, *task completion time*, consists of five phases, and the time spent on each phase is, generally, independent. The time required for each phase is known as: *service location time*, *task allocation time*, *waiting for capacity time*, *task execution time*, and *task return time*, each of which can be seen in Figure 3.3, and is described below.

On two occasions during the completion of task t in Section 3.3, a service needs to be located. The amount of time required to locate a service is known as *service location time*, and is based on the cost of communicating with other agents when locating a service. *Service location time* can vary greatly according to how the system is designed, and how it operates. The most obvious factor that can affect *service location time* is the approach used to locate services. Once an agent offering task t's required service has been located, t is allocated. The time needed to allocate a task is called the *task allocation time*, and like *service location time*, is based on the cost of communication.

When an agent receives a task that it can execute, the next step is to execute it, but not until the agent has the capacity to do so. The amount of time between a task being received and execution starting is called *waiting for capacity time*. If an agent has already received more tasks than it can execute simultaneously, then *waiting for capacity time* for task t will be above zero. If the system as a whole is overloaded, or an agent receives a disproportionate allocation of tasks, then *waiting for capacity time* may be very high. When the system as a whole is overloaded we cannot remove the problem, but aim instead to handle it gracefully by beginning to execute waiting tasks as soon as possible. In the latter case when an agent receives more tasks than others, an alternative allocation approach can be used to direct tasks elsewhere; in addition, if an agent receives a few very large tasks, then it may appear more busy than another with many small tasks.

Once an agent has capacity to do so, it starts executing the task. The actual time spent on execution is *task execution time*, and is determined by the requirements of a task. For

simplicity, we assume that all agents execute tasks at the same speed, so task execution time cannot be altered or affected. Finally, once execution is finished, the results must be returned to agent that initially received the task. The time taken for task results to be returned is called *task return time*, which is the same as *task allocation time*, because it is the same process in reverse. Each of these *phases* contributes to the overall time taken to complete a task. If the task is complex (and has multiple subtasks), then the total is not simply the sum of all these parts, since some subtasks can be executed concurrently. However, the time to complete an atomic task is the sum of all these values.

$$Task\ Completion\ Time = \sum \left\{ \begin{array}{c} Service\ Location\ Time \\ Task\ Allocation\ Time \\ Waiting\ for\ Capacity\ Time \\ Task\ Execution\ Time \\ Task\ Return\ Time \end{array} \right\}$$

3.6 Communication

Communication between agents is a large component of completing a task. Indeed, of the five phases of *task completion time* discussed in Section 3.5, *service location time*, *task allocation time*, and *task return time* are all built up from the cost of communication between agents.

The cost of communication is derived from the cost of the messages that must be sent back and forth between agents when communicating. This can be broken down into two factors: the computation needed to send a message; and the time taken for a message to be sent from one agent to another. However, compared to the computational cost of executing tasks, the computational cost of compiling and sending a message is negligible. From a telecommunications network perspective, the time to send a message from one agent to another is a complex issue involving latency and bandwidth, so we instead abstract away from it as follows: if a message is sent from an agent a_1 to an agent a_2, and a_1 transmits the message at time n, then the message will arrive at agent a_2 at time $n + 1$.

While various approaches may be used to reduce the number of messages required to locate services and allocate tasks, such as various mechanisms introduced by Didona et al. [33], our concern is not with the absolute number of messages sent or received, but rather the relative number of messages sent when comparing two alternative systems.

We therefore appeal to the principle of *monotonicity*, that any mechanism used to reduce communication in a particular system, reduces communication costs in all systems in the same way, and magnitude. As this will not affect the relative communication cost between systems, such mechanisms are not considered.

3.7 Conclusion

In this chapter we have elaborated the eScience scenario from Section 1.3, one of many possible situations in which organisational adaptation can be beneficial. The scenario has been formalised as a *task allocation and execution model*, which specifies how tasks are built up from the requirement for services and the internal structure of agents that execute these tasks.

Moreover, we have presented an analysis of *task completion time*, which is made up of five phases, and which varies depending on how services are located. Interestingly, changing the design and operation of a system does not affect all phases of task completion. Clearly, *task execution time* is determined by the underlying speeds of devices, but for simplicity we assume that all agents operate at the same rate, and do not consider this further. We can conceive that *task allocation time* and *task return time* can be affected by changing how services are located. For example, if one approach allocates a task to the first agent located that offers a required service, the requirement will be satisfied, and the next required service must be located. Alternatively, if a second approach instead searches until multiple agents are located, and then allocates the task to an agent that can satisfy multiple service requirements, rather than just one, then fewer *task allocations* are needed and, by inference, fewer tasks need to be returned. However, to achieve this, the service location process must locate multiple agents, which implies that when one agent has been located, the search process continues. Though this can decrease *task allocation time* and *task return time* slightly, *service location time* is typically increased significantly as a result, countering the benefit.

Thus, to summarise, in the context of our *task allocation and execution* model, task throughput, or *task execution time*, can be affected by the approach used to locate services. However, this generates minimal impact on *task allocation time* and *task return time*. Instead, any change to the service location approach only affects *service location time* and *waiting for capacity time*, and this is where we focus in the remainder of this thesis through the next few chapters.

Chapter 4

Evaluating Task Throughput

4.1 Introduction

As we have discussed previously, our concern in this thesis is to study task execution and allocation in distributed multi-agent systems. Our approach is experimental: in seeking to understand the underlying model and its consequences, and in seeking to develop mechanisms to improve task allocation and execution, we require a simulation environment with which to examine these issues. In this chapter, therefore, we describe just such a simulation environment to evaluate the performance of our *task allocation and execution model*, and to be used as a base for evaluating subsequent developments. Importantly, we also consider how to evaluate, and provide metrics that will be used subsequently as a means of determining the performance of different mechanisms proposed in this thesis. Since the model has already been provided, we need only consider here how that model is instantiated, through the generation of tasks and agents, and the metrics themselves.

This chapter is structured as follows. Section 4.2 and Section 4.3 describe how to generate tasks and agents respectively, both of which were formally introduced in Chapter 3. In Section 4.4 parameters are introduced for our simulations, and in Section 4.5 metrics are introduced to evaluate the performance of task throughput based on *task completion time*.

4.2 Tasks

According to our *task allocation and execution model*, a task takes the form $t = (\text{R}, \text{ROT})$, consisting of a set of *requirements*, R, and a *requirement ordering tree*, ROT, which specifies the order in which these requirements must be satisfied. We describe the process that is used to generate a task in two parts, first describing the process to generate *requirements*, and then the set of *ordering constraints*.

4.2.1 Requirements

As stated previously, a requirement specifies the need for a service s for an amount of time srt, and so is a tuple $r = (s, srt)$. To generate a requirement in our simulation environment, s is instantiated as a random service from the set of all services, S. Because srt cannot be infinite, and must be specified, it is instantiated as a value between srt_{min} and srt_{max}, according to a linear distribution. When a task simply has a single requirement, this is all that is required, but when there are multiple requirements, the set of requirements must be generated differently, as follows.

Recall that our work is predicated on the assumption of a correlation between services, by which the choice of, or need for, one service in a task influences the choice of others. In practical terms, this *requirement correlation* demands that all requirements in a single task will only require services from a subset of services called a *service category*, SC, so the generation of a requirement is parametrised by the *service category* from which it can require services $\text{SC} \subset \text{S}$. The required service is an element $s \in \text{SC}$, and is randomly selected using a linear distribution across all the elements of SC. The value srt is generated as specified above.

Since requirements cannot themselves be infinite, the set R for each task has between nor_{min} and nor_{max} number of requirements, where the number of requirements is determined using a linear distribution across this range. Once the number of requirements is determined, the prescribed number of requirements are generated and added to R.

4.2.2 Ordering Constraints

Now that we have a set of requirements, we can generate a *requirement ordering tree* to specify the order in which these requirements must be satisfied. A task has a set of

ordering constraints of the form (r_x, r_y) that collectively form the *requirement ordering tree*, ROT.

To create a *requirement ordering tree*, first a *requirement* is chosen to be the root of the tree. An arbitrary number of dependants are then selected from the set of remaining requirements, and a dependency is created between the root requirement and each of the selected requirements. Each of the selected requirements is then in turn assigned a number of dependants. This process is repeated until all requirements are linked together, and in this way a *requirement ordering tree* is generated.

More formally, Algorithm 1 shows how a *requirement ordering tree* can be generated for a set of *requirements*, R. First, all but one of the requirements is placed in set, NAD: a set of requirements, each of which has not yet been linked to the requirement ordering tree, or is *not a dependant* (lines 1–2). The remaining requirement r' is placed in set, LWD: a set of requirements that are linked to the requirement ordering tree, but have no dependants of their own, or are *linked, without dependants* (line 3). On line 4 we start to iterate over each requirement in set LWD, to attach dependants. Initially, LWD only contains r', but additional requirements are added as they are connected (line 14). Line 5 checks if any requirements are not yet linked to the requirement ordering tree. If NAD is empty, then the ordering tree is complete. If not then line 7 specifies the number of dependants, *nod*, for the current *parent requirement*, *pr*. This is a random value between nod_{min} and nod_{max}. If the number of dependants is greater than the number of remaining unconnected requirements (line 8), then *nod* is reset to the size of the set NAD on line 9. Finally, requirements are iteratively removed from the NAD, each of which is called a *dependant requirement*, *dr*, and a dependency is created between *pr* and *dr* on lines 11–13. Now that *dr* is connected to the tree it can be added to LWD, and *nod* is decreased. This continues until all requirements are connected, at which point ROT is populated with *ordering dependencies* that form a *requirement ordering tree*, and so is returned.

4.3 Agents

Recall from Chapter 3 that an agent takes the form $a = (\text{SI}, \text{T}, \text{RT}, rtc, \text{C})$, where SI is the set of service instances that agent a offers, T is the ordered set of received tasks, RT is the set of running tasks, *rtc* is an agent's *running tasks capacity*, and C is a set of connections, through which a is connected to other agents. At runtime, agents receive

Algorithm 1 *generateTree*(R, nod_{min}, nod_{max})

```
1.  for r ∈ R \ {r'} do
2.      NAD ← NAD ∪ {r}
3.  LWD ← {r'}
4.  for pr ∈ LWD do
5.      if |NAD| = 0 then
6.          break
7.      nod ← random(nod_min, nod_max)
8.      if nod > |NAD| then
9.          nod ← |NAD|
10.     while nod > 0 do
11.         dr ← r | r ∈ NAD
12.         NAD ← NAD \ {dr}
13.         ROT ← ROT ∪ {(pr, dr)}
14.         LWD ← LWD ∪ {dr}
15.         nod ← nod − 1
16. return ROT
```

tasks and place them in set T, which is initially empty. Likewise, RT is populated at runtime when an agent is executing a task, so it is initially empty. Thus, to generate an agent according to our model, only SI, *rtc*, and C need to be initialised.

All agents offer a fixed number of service instances, represented as *nsi*. These services are therefore selected from the set S, and a service instance is added to SI for each. Every agent's *running task capacity* value, *rtc*, is fixed for all agents. We can imagine situations in which this value might vary across agents, indicating that agents have different resources, and therefore higher or lower capacity for executing tasks, but for simplicity we remove this from consideration.

Finally, we must generate the set of connections that link all agents in a system. Between every possible pair of agents exists a binary relationship, such that there is a connection between them, or there is no connection, so for a set of n agents, there are $\frac{n\times(n-1)}{2}$ pairs of agents that can be connected or, put more succinctly, $\binom{n}{2}$ pairs. For all these agents, there are $2^{\binom{n}{2}}$ possible set of connections that link them. So, for example, if a system contains 100 agents, then there are 4950 agent pairs, and approximately 1.25×10^{1490} possible structures. Naturally, this is a large number of structures that is both infeasible, and unrealistic to consider, but the set of connections is often restricted depending on, for example, the approach used to locate services. Therefore, we instead instantiate the connections between agents as they are needed in later evaluations.

4.4 Simulation and Parameters

This section specifies the default parameters used for all experiements in the remainder of this thesis, unless otherwise stated for a particular experiment. In undertaking simulations themselves, each simulation consists of 100 simulated agents that collectively receive a mean of 10 tasks at each time step, where the actual number arriving at each point is dictated by a Poisson distribution. Tasks are initially allocated to agents according to a linear distribution, and all agents execute tasks at the same rate. Each agent offers two service instances. The initial parameters for each simulation are summarised in Table 4.1.

Using the description of all key elements above, we implemented a simulator for this agent based system, to perform various experiments. The simulator was developed using the Java language and libraries, and some additional libraries were used, including the Java Universal Network/Graph (JUNG) Framework[1] to implement connections between agents and for various graph based algorithms, the Colt library[2] to generate random numbers for Poisson and Normal distributions, and JUnit[3] for testing. Each experiment was performed on a machine containing 8GB of RAM, and a CPU running at 1.9GHz.

Parameter	Description	Default Value
srt_{min}	The minimum value for service requirement time	1
srt_{max}	The maximum value for service requirement time	20
nor_{min}	A task's minimum number of requirements	1
nor_{max}	A task's maximum number of requirements	10
nod_{min}	Minimum number of dependencies for one requirement	1
nod_{max}	Maximum number of dependencies for one requirement	4
nsi	Number of service instances offered by each agent	2
ns	Number of services, of which agents offer instances	200
nsc	Number of service categories	25
rtc	Agent capacity for simultaneous running tasks	10

TABLE 4.1: Default parameters for all agent simulations

[1] http://jung.sourceforge.net/
[2] http://acs.lbl.gov/software/colt/
[3] http://junit.org/

4.5 Evaluation Metrics

Now that we have specified the details of our simulation environment, we can proceed to a consideration of the metrics that are needed in order to analyse the task throughput achieved by a particular configuration. Each experiment consists of the same simulation being repeated 25 times, with the results averaged over all runs. Based on the *task completion process* described in Chapter 3, *service location time* and *waiting for capacity time* are individually assessed in each experiment.

The first part of the evaluation assesses *service location time*. The average *service location time* is evaluated at each time step, to determine whether *service location time*, as a whole, is improved. However, over the course of a simulation, thousands of services need to be located. Average *service location time* can suggest a trend towards improved performance, but it can also hide outliers. Therefore, the frequency distribution of *service location time* is also assessed to determine whether outliers exist and, if so, how regularly they occur.

The second part evaluates the additional effects of structural adaptation and, in particular, the effect of changing *service location time*. As we have said, task completion consists of five phases, each of which is a distinct, sequential part of the task completion process, so changing one phase can have an effect on all subsequent phases. For example, if the rate at which services are located changes, then the rate of task allocation will also change, which will affect the load of all agents receiving tasks. We use *waiting for capacity time*, *agent load*, and the number of overloaded agents, to assess any by-products of changing *service location time*.

At various points in our evaluations, statistically significance is used to validate our results. When significance is mentioned throughout the remainder of this thesis, this is based on the results from a T-test, where a resulting p-value ≤ 0.05 is considered significant, and a p-value ≤ 0.01 is considered highly significant.

Chapter 5

Centralised Service Location

5.1 Introduction

As we have seen, one of the most crucial aspects of task allocation lies in the time taken to locate relevant services and the agents that offer them. Perhaps the most obvious and most simple means of locating services is to use a central registry to maintain information on individual agents and services. This is the focus of this chapter, which describes and analyses the performance of three different approaches that locate services in a centralised manner.

In particular, we introduce approaches to locate services using a centralised registry, with the registry providing different functionality in each approach. We do not claim any contribution in the development of these approaches, which follow the client-server model that is commonplace in many distributed systems [11]. Instead, we aim for two things. First, we aim to provide a baseline for evaluating performance of different task allocation mechanisms as a means of comparison with subsequent developments. Second, we aim to provide a detailed analysis of each of these approaches on the system as a whole. As discussed earlier, both *service location time* and *waiting for capacity time* can be affected by the service location approach used, and we examine the impact on each.

This chapter is structured as follows. In Section 5.2 three *service location approaches* are introduced, and in Section 5.3 these approaches are evaluated and the results analysed. The chapter is concluded in Section 5.4.

5.2 Service Location

When analysing the tasks it has received, an agent will likely come across a task that it cannot execute by itself because it does not offer one or all of the services that the task requires. In this situation an agent must execute as much of the task as it can, before allocating it to another agent that can satisfy the next service requirement. However, before the task can be allocated, first another agent must be found that offers an instance of the required service.

In this section we focus on centralised methods to locate services. Each approach described here is considered centralised because all service location is brokered through a *central registry* that stores different information about the agents and services within the broader system framework. As we have seen, agent interaction is guided by the relationships or *connections* between agents, and different *connections* can thus determine the performance of a particular system. In the case of *centralised service location*, agents locating services always begin the search process by communicating with the central registry. Therefore, on initialisation, each agent is connected to this central registry, and no other connections exist.

5.2.1 Central Agent Registry: Isolated Requests

The problem of locating services can be simply solved by using a *central agent registry* that maintains contact data on all agents in the system; it can then be queried to find a single random agent. If an agent k needs to find another agent offering service s, k first sends a request to the registry, and the registry replies with details of a random agent m_1. Agent k then asks m_1 if it offers service s, and m_1 replies with confirmation if it does, at which point k allocates the task to m_1. If it does not offer the service, then it replies with a denial, at which point k requests from the registry another agent's details, m_2. This *centralised search* is very simple, and ultimately does not fail to locate services (assuming no communication problems). However, since the registry does not track requests, it is possible that $m_1 = m_2$, and so multiple denied requests to the same agent are also possible. Because the *registry* does not consider the result of previous requests when responding to a query, we say that this approach uses *isolated requests*.

5.2.2 Central Agent Registry: Sessions

As a development of the service location approach that sends *isolated requests*, the *central agent registry* might instead record previous requests from each agent. Here, it can more effectively respond in the future by excluding responses that each agent has previously received. For example, in the case above this would ensure that $m_2 \neq m_1$. In this way, every time an agent needs to locate a service, the registry maintains a service location *session* for the agent until the service is found. Once the service is found, the session ends, and the information recorded is deleted. The next time an agent wants to locate a service a new session will begin. However by introducing sessions for each agent, a higher demand on computation and memory is also introduced, putting more strain on the central registry that already represents a significant communication bottleneck. Depending on the number of agents in the system, and the numbers of services and requests, this could be excessive.

5.2.3 Central Service Registry

The final *centralised location approach* goes one step further, such that the central *registry* does not just track each agent, but also the services offered by each agent, in a similar fashion to the *matchmaker* concept introduced by Ben-Ami and Shehory [8]. We call this the *central service registry*. This converts the central registry into a complete black box, such that an agent can request an instance of a particular service, and the registry responds with the identity of an agent that offers the required service. When the central registry receives a request, it checks its records for an agent that offers the required service. Of all the agents that offer the required service, one agent is selected, and its identity is returned to the requesting agent.

As the *central service registry* locally stores information about all services, the location process is significantly faster, since only two messages need to be sent to locate a service (taking two time steps). However the central registry's computational load and communication costs increase significantly compared to the previous two centralised approaches because the registry must store potentially large amounts of information about each agent. When the *central service registry* is queried, it must search through this large data store, and if the system's agent population is dynamic then the *central service registry* must expend time and effort in ensuring that its information on agents

and services is kept up-to-date. Such additional pressure on the *central registry* is not apparent when considering service location time.

5.3 Evaluation

Given our description of these three centralised *service location* approaches, we undertook experiments to analyse the performance of each, in the context of the simulation environment and experimental set-up as described in Chapter 4, and the experimental parameters described in Section 4.4. The first two approaches each make use of a *central agent registry*, where the first uses *isolated requests*, and the second maintains *sessions* for each agent. The third approach makes use of a *central service registry*. Each of these approaches operates on top of a static organisational structure, where each agent has a connection to the central registry. More formally, and with respect to the *task allocation and execution model* introduced in Chapter 3, for a multiagent system $mas = (\text{A}, \text{C})$, where $\text{A} = \{(a_1, a_2, \ldots, a_n)\}$, all agents are connected to a *central registry*, *cr*, as follows: $\text{C} = \{(a_1, cr), (a_2, cr), \ldots, (a_n, cr)\}$. For every agent $a_i \in \text{A}$, such that $a_i = (\text{SI}, \text{T}, \text{RT}, rtc, \text{C}_i)$, the agent a_i has one connection as follows: $\text{C}_i = \{(a_i, cr)\}$.

5.3.1 Central Agent Registry: Isolated Requests

Our first experiment evaluated *task completion time* when locating services by making *isolated requests* to a *central agent registry*. For this approach, an average *service location time* of 400 time steps is achieved, as shown in Figure 5.1, but service location can — on rare occasions — take up to nearly ten times this. This is shown by the frequency distribution of *service location time* in Figure 5.2(a), which highlights a tail-off effect such that services are generally located quickly, but on rare occasions service location takes a significantly larger amount of time. The bars in the graph indicate the number of instances for which service location took a value in a limited range. The graph is shown as a histogram for ease of inspection, and the results can be understood probabilistically, as follows.

Suppose we have a set of agents A, only one of which offers the required service. A request is sent to the central registry, and the identity of an agent is returned. Now, the probability of the returned agent offering the required service is $\frac{1}{|\text{A}|}$, and conversely, the probability of the agent not offering the service is $\frac{|\text{A}|-1}{|\text{A}|}$. If the service is not found,

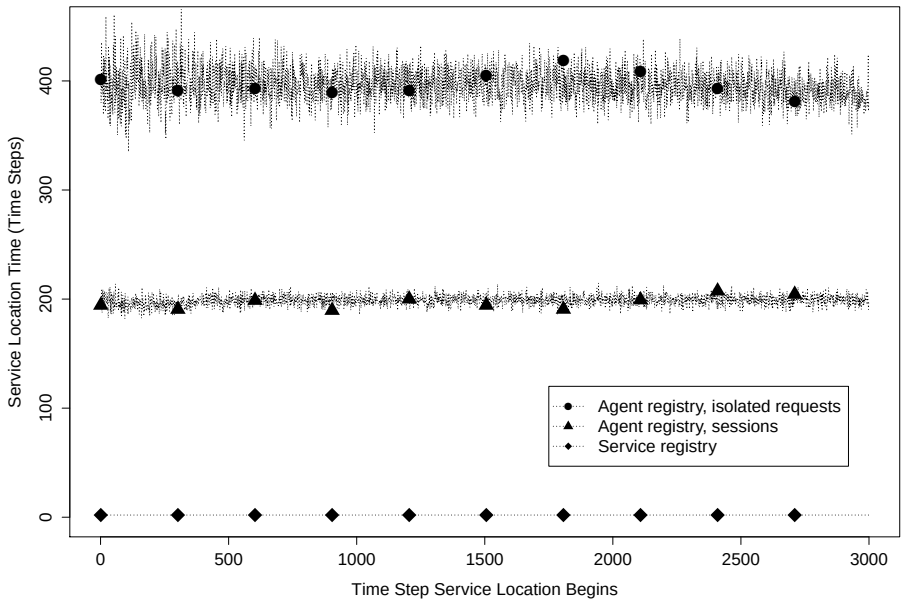
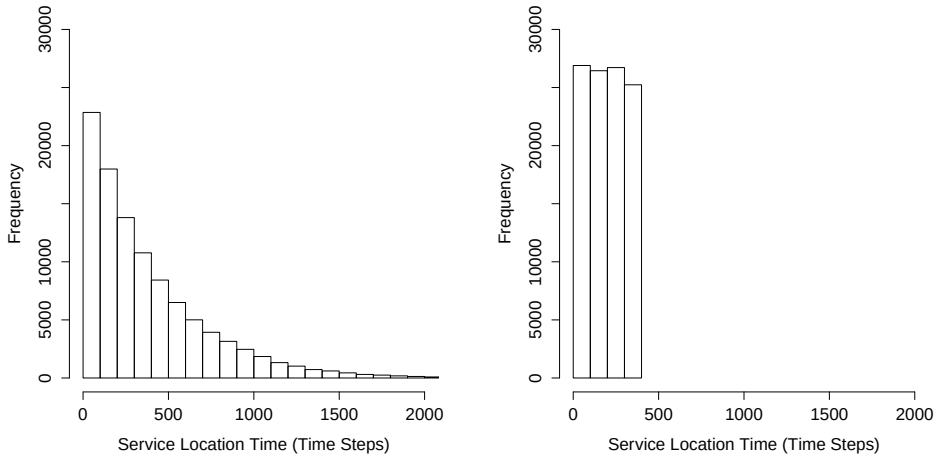


FIGURE 5.1: Centralised service location: service location time



(a) Centralised, isolated requests

(b) Centralised, sessions

FIGURE 5.2: Centralised service location: service location time frequency distribution

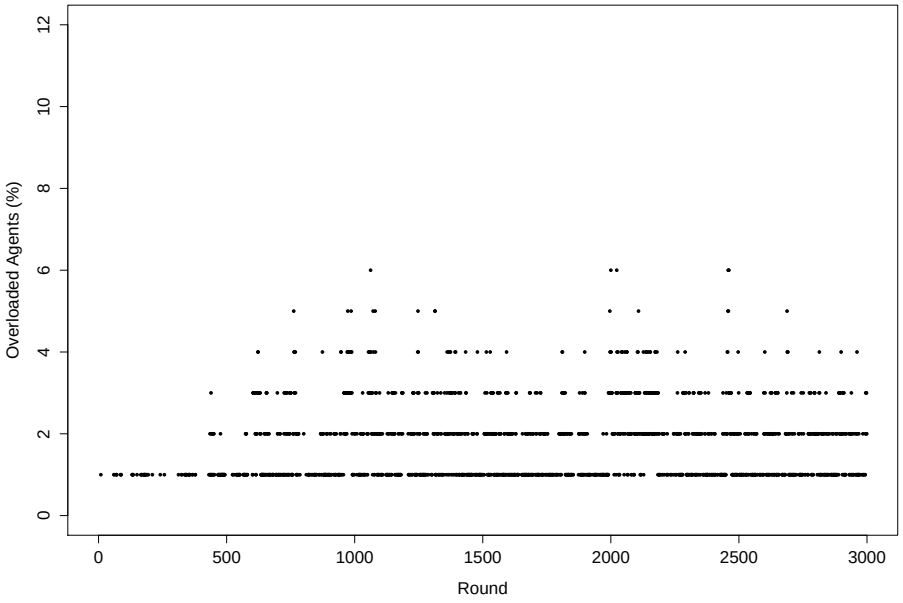


FIGURE 5.3: Centralised agent registry, isolated requests: overloaded agents

then a second request is sent and, because each request is isolated from all others, the probability of the second agent offering the service is again $\frac{1}{|\mathrm{A}|}$, and the probability of the second agent not offering the service is $\frac{|\mathrm{A}|-1}{|\mathrm{A}|}$.

Given this, we can infer that the probability the service has not been located after making n requests is thus equal to

$$\left(\frac{|\mathrm{A}|-1}{|\mathrm{A}|}\right)^n$$

so that the probability that the service is not located after making n requests is always positive (though it becomes small very quickly). For example, if there are 100 agents, the probability of not locating a service after making 500 requests drops to 0.007. The probability of not locating the service after making 5000 requests is approximately 1.5×10^{-22}: possible, but very unlikely. The tail-off effect makes this approach clearly less desirable than one that can guarantee service location after a fixed period.

By examining the number of overloaded agents in Figure 5.3, we can see that a small number of agents are overloaded, and Figure 5.4 shows that the average load is approximately 45-50%. Figure 5.5 shows that average *waiting for capacity time* is never high,

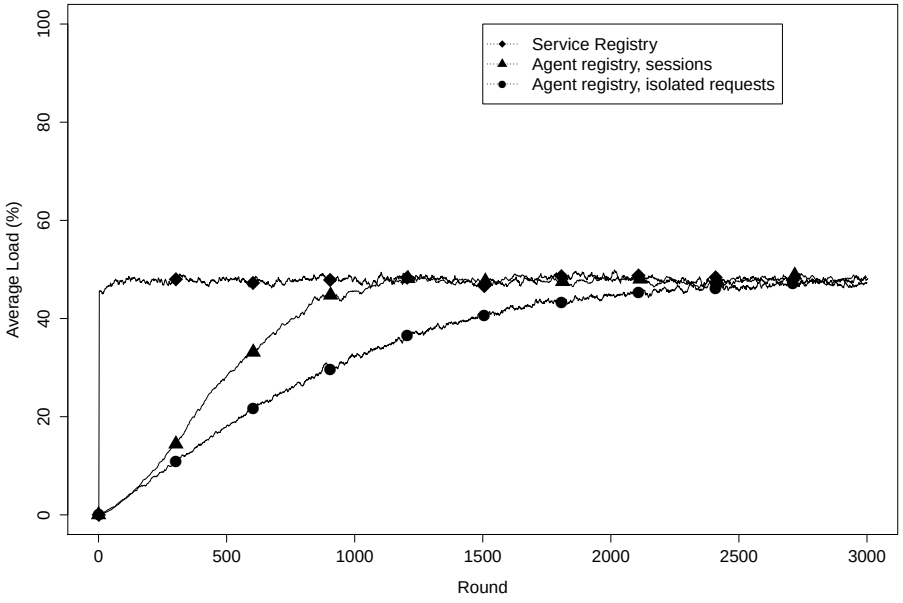


FIGURE 5.4: Centralised Service Location: Average Load

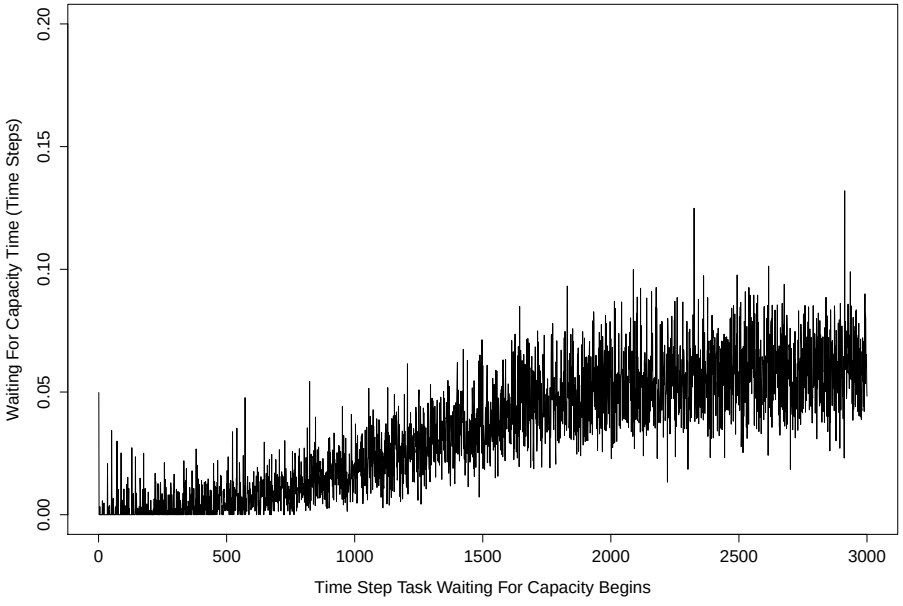


FIGURE 5.5: Centralised agent registry, isolated requests: waiting for capacity time

though tasks must, at times, wait to be executed, and the number of rounds they need to wait initially increases slowly and then levels out. In the worst case, approximately one in 10 tasks must wait a single round before it can be executed. In principle, *waiting for capacity time* can be improved by ensuring load is distributed better. However, in the context of our experiments there is only one instance of each service, and so load cannot be distributed.

The gradual rise in average load displayed in Figure 5.4 can be attributed to the very high *service location time* shown in Figure 5.1 and wide frequency distribution shown in Figure 5.2(a). Initially no tasks are being executed, so the average agent load is zero. Once service location begins, a large proportion of the required services are located quickly, so tasks can be allocated and executed, causing the average load to rise. Finally, as shown by the tail-off effect in the service location frequency distribution, a minority of services take a long time to be located, so load continues to rise but at a slower rate.

In summary, using *isolated requests* with a *central agent registry* will always successfully locate services. However, as demonstrated by the tail-off in the distribution of *service location time*, the amount of time needed to locate a service is extremely unpredictable. In addition, few tasks have to wait to be executed, and when they do their wait is short, but many agents are idle suggesting that load distribution can be improved.

5.3.2 Central Agent Registry: Sessions

Our second experiment evaluates *task completion time* when locating services through a *central agent registry* that maintains *sessions* for each service location process. By remembering the agents that have already been contacted during a particular search process the tail-off effect, present in the previous service location approach, can be removed. In fact, in doing so, this makes *service location time* much more predictable.

The time required to check if one agent offers a particular service is four time steps (a request and reply to the central registry, plus a request and reply to the agent identified). Thus, for a system of 100 agents, the time to locate a service is always somewhere between 0 and 400 time steps, as shown in Figure 5.2(b), and the average time needed to locate a service is 200 time steps, as shown in Figure 5.1. More generally, for a system that consists of n agents, *service location time* is, in the worst case, $4n$ and, on average, $2n$. This performs better than *isolated requests*, but requires the registry to keep a

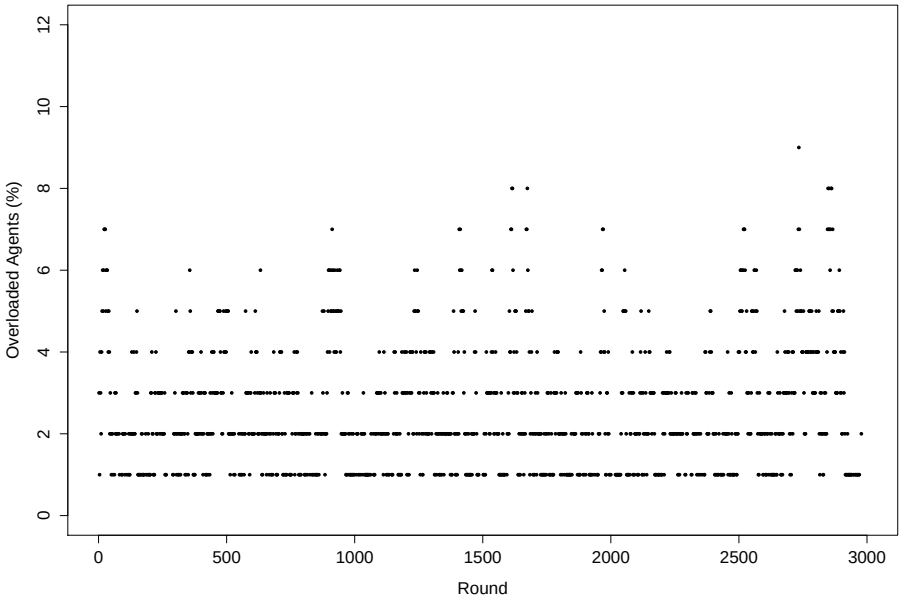


FIGURE 5.6: Centralised agent registry, sessions: overloaded agents

record of the progress of each service location process for every agent. It is thus more computationally and memory intensive for the central registry.

In addition, load can be affected by the rate of task allocation, and when *sessions* are maintained by the registry, required services, and therefore required agents, can be located more quickly. For this reason, the results achieved for agent load in Figure 5.4, the number of overloaded agents in Figure 5.6, and *waiting for capacity time* in Figure 5.7, are the same as for *isolated requests*, with the single difference that any change to performance occurs more quickly, such as the initial rise in agent load.

To summarise, by utilising a *central agent registry* that maintains *sessions* for each service location process, tasks can now be located faster, and within a well understood upper bound. The distribution of load is also similar, when compare to *isolated requests*. However, this requires that the *central registry* performs more work.

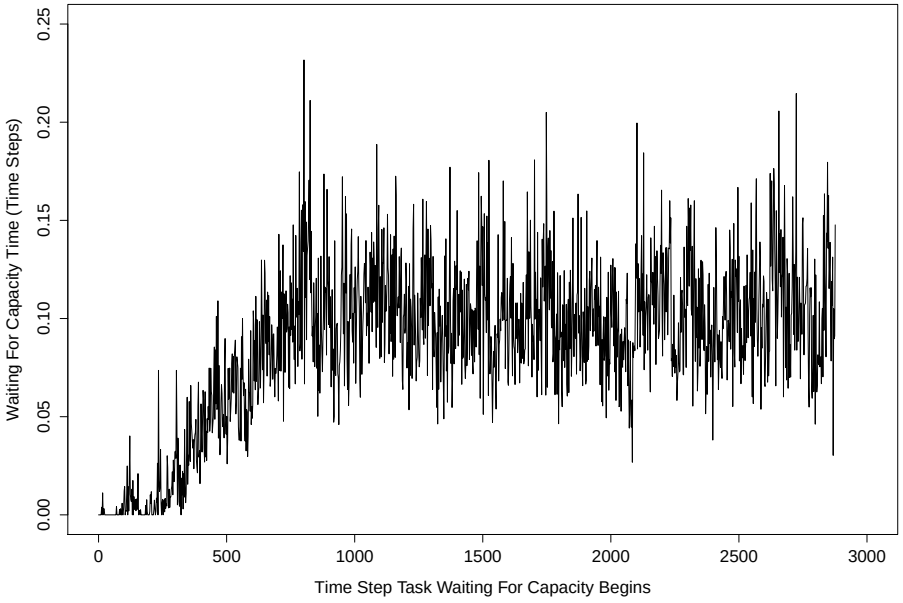


FIGURE 5.7: Centralised agent registry, sessions: waiting for capacity time

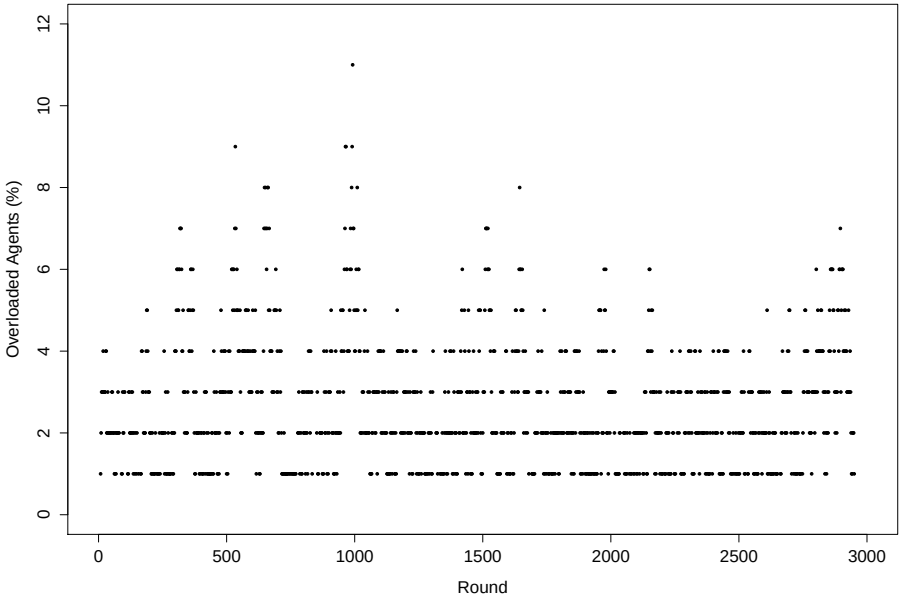


FIGURE 5.8: Centralised service registry: overloaded agents

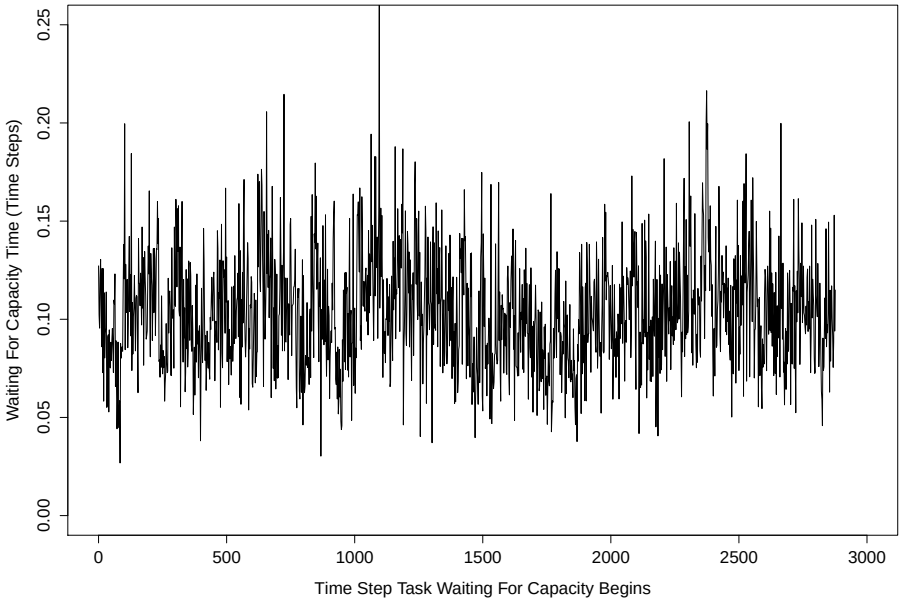


FIGURE 5.9: Centralised service registry: waiting for capacity time

5.3.3 Central Service Registry

Our final experiment in this chapter evaluates *task completion time* when using a *centralised service registry* to locate services. Because the central registry maintains information about all agents, and the services they offer, it can return the identity of an agent that is guaranteed to offer the requested service. In addition, because this information is local, *service location time* is consistently two time steps, as shown in Figure 5.1, and as all services are located in the same amount of time, there is no distribution to show. The tasks that require services can also be allocated much earlier so, as before, all changes in performance occur more quickly, though ultimately all results in this respect are the same as *isolated requests* and *sessions* (agent load, Figure 5.4; number of overloaded agents, Figure 5.8; *waiting for capacity time*, Figure 5.9).

However, this approach requires a significant amount of work from the central registry that is not immediately apparent when evaluating *service location time*, and *waiting for capacity time*. First, storing information about each agent in the system can require a large amount of memory depending on the number of agents in the system, and the number of services they offer. Second, ensuring that this information is up-to-date requires constant maintenance. Though this approach can achieve extremely low *service*

location time, the extra load placed on the central registry to achieve this increases the problem of the registry being a significant bottleneck.

5.4 Conclusion

In this chapter we have introduced three approaches to locate services in a distributed manner. The first two approaches rely on a *central agent registry* that maintains a list of each agent in the system, and the third approach makes use of a *central service registry* that maintains information about each agent, and each service instance it offers.

Through evaluating these approaches we have shown that *service location time* is worst when the *central registry* treats every request in isolation. This also produces unpredictable *service location times*, such that services are usually located in a reasonable amount of time, but can occasionally take a significant amount of time to be located. *Service location time* can be decreased when the central registry maintains *sessions*, and this also introduces an upper limit so that *service location time* never takes longer than 400 time steps (the average *service location time* for *isolated requests*). However this requires the central registry to do more work. By using a central *service* registry instead of a central *agent* registry, service location time can be significantly improved. However, to achieve this the *central registry* must store information about each agent, and the services they offer, and also constantly maintain this information to ensure that it is up-to-date. The *waiting for capacity time* achieved by each of these approaches is similar, the single difference being that changes in *waiting for capacity time* occur at different rates depending on *service location time*.

To conclude, across these three approaches there is a trade-off between *service location time* and the load on the *central registry*. When using *isolated requests*, the load on the central registry is low, but this produces the worst results with respect to *service location time*. Using a *central service registry* achieves far superior results with respect to *service location time*, but as the registry needs to maintain a significant amount of information, the load is significantly higher. Finally, regardless of approach, each relies on a central entity to locate services, introducing a single point of failure. Clearly, in a distributed system this is suboptimal, and needs to be addressed. In what follows, therefore, we consider how to remove both the bottleneck and single point of failure that this central registry represents so that *service location time* can be improved without having to trade-off one aspect against another.

Chapter 6

Decentralised Service Location

6.1 Introduction

As we have seen, using a central registry to facilitate service location can be applicable in some contexts, but to locate services in a timely manner, the central registry is placed under significant load, limiting the scale at which centralisation is applicable. In consequence, for services to be effectively located at any scale, agents must be capable of locating services in a decentralised manner. Now, when locating services, agents communicate across the organisational structure by which they are connected. In the case of a central registry, this structure can only take one form, as seen previously. However, when services are located in a decentralised manner, this structure can vary, with different structures potentially impacting on an agent's ability to locate services in different ways. This is the focus of this chapter, which describes and analyses the performance of a single decentralised service location approach, across four different organisational structures.

The chapter is structured as follows. In Section 6.2 we discuss the categorisation of organisational structures according to any structural constraints, and describe four typical organisational structures over which decentralised service location might operate. In section 6.3 we introduce an approach to locate services in a decentralised manner, effectively removing the system's reliance on a central registry. In Section 6.4 we analyse the potential effects of each structure on the performance of service location, and then in Section 6.5 we provide an empirical evaluation. Section 6.6 concludes.

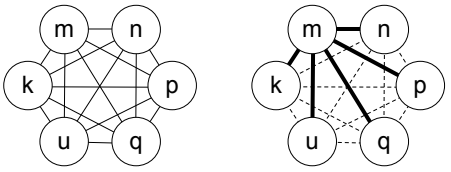


FIGURE 6.1: Fully connected structure: connections linking m to all other agents

6.2 Organisational Structures

When a central registry is used, the organisational structure over which agents communicate is constrained, such that all agents indirectly locate each other through the central registry. As discussed, this causes a bottleneck and a single point of failure. To address this, we might consider decentralising service location by which this central registry is removed, and services must instead be located though direct communication between agents that require services and agents that offer services (and, depending on the context, agents can be on both sides of this interaction). However, because agents communicate across the set of all connections that link them — across the *organisational structure* — these links are crucial. In particular, in the centralised approach, the connections exist only between the central registry and each agent in the system. In order to decentralise, therefore, we must ensure that there are appropriate connections between the agents in the system to a sufficient degree that decentralised service location is facilitated. A simple solution is to create a connection between every pair of agents, as shown on the left in Figure 6.1, so that each agent can communicate with all others directly. For example, the same structure is displayed on the right hand side of Figure 6.1, and highlights the connections, in bold, that link agent m directly to every other agent.

However, when a system is of any significant scale this becomes, at some times, impractical because it is expensive to maintain the amount of information required and, at others, infeasible because it is not possible for every agent to keep an up-to-date record of every other agent in the system. Instead, we must consider alternative structures where the number of required connections is reduced, but services can still be located.

Our agents are, to a degree, cooperative, and so will help each other to locate services, so we do not need a *fully connected structure*. To ensure that any agent can locate a service offered by any other agent, we do not require a *connection* between each pair of agents, but rather a *path* of connections between every pair. This is to say that, if the

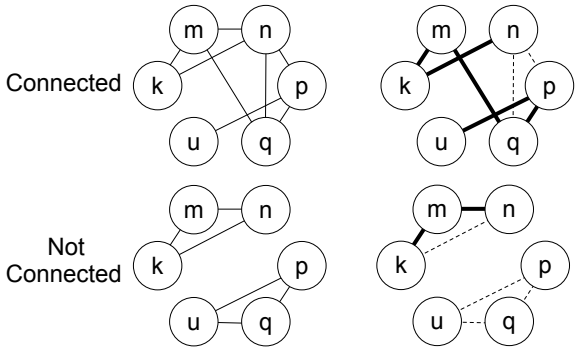


FIGURE 6.2: Path of connections between agent m and all other agents

organisational structure is a graph, where agents are vertices, and connections between agents are edges between these vertices, then the graph must be *connected* such that the graph is not partitioned into multiple subgraphs. An example of this can be seen in Figure 6.2, where the two top structures are connected, and the two bottom structures are not connected, because they are divided into subgraphs. At the top left of the figure, the structure is not fully connected, yet at the top right, a path of connections exists between agent m, and every other agent (and these connections are shown in bold). At the bottom left of the figure, a path of connections exists between agent m, and agents n and k, but no path exists between agent m, and agents u, p, and q. In summary, to ensure that every agent can locate every other agent, the organisational structure does not need to be *fully connected*, but it must be *connected*.

Such connected structures can be commonly seen in many kinds of distributed systems, both computational and natural, and are typically referred to as *topologies*, which can be broadly divided into two categories according to the context in which they appear. The first category refers to patterns that emerge in complex systems, such as societies and biological systems. These are also referred to as *complex networks*, and the topology is not intentionally created or constructed, but instead these patterns emerge. Work by Kittock [63] falls into this category, where a lattice is used to simulate restricted interaction between agents. Similarly, Newman [85] discusses the properties of a number of *complex networks*, such as scale-free networks.

The second category refers to topologies that are specifically constructed or designed for their beneficial qualities, such as hierarchies or pipelines, which offer structure that serves to regulate or organise the interactions between entities. In Section 1.2 we introduced a *telecommunications network*, which is an example of those that are specifically designed

to follow a particular organisational structure, in this case a hierarchy. These kinds of structure are the focus of this thesis, since we are interested in structures that are designed and maintained, and therefore consider topologies that fall into this category.

The topologies we consider are drawn from physical layer communications networks, which are usually categorised as *point-to-point*, *bus*, *line*, *ring*, *star*, *tree*, *mesh*, and *fully connected* networks. However, as we will see, we need only consider *line*, *tree*, *mesh*, and *fully connected* networks, since all others can be reduced to these four. We first discuss these four topologies, and then discuss the remaining topologies showing how each is reduced to one of our four.

A network following a *line* topology consists of a collection of entities that are connected to each other sequentially, as shown in Figure 6.3(a), such that all entities have two connections, apart from the two entities at each end of the line that have one connection. A network that follows a *tree* topology is connected via superior-subordinate relationships, shown in Figure 6.3(b). Every entity is a subordinate to one entity, and can be a superior to many other entities. There is a single entity that is an exception to this rule at the top of the tree, with no superior. A network that follows a *mesh* topology has any number of connections that generally have no clear structural constraints, as in Figure 6.3(c), and so the connections are arbitrary. When a network follows a *fully connected* topology, every entity has a direct connection to every other entity, as shown in Figure 6.3(d).

The remaining four topologies can be reduced to one of these four. A *bus* network is technologically very different to a *line* network at the hardware level. However, since the networks that we consider are at the virtual level, these two topologies are similar enough to be considered the same. A *ring* network is essentially a *line* network that loops back on itself much like a circular queue, and is sufficiently similar. A *star* network is a specialised *tree* that has a limited depth of two. Finally, a *point-to-point* network is simply a *line* network with a length of one.

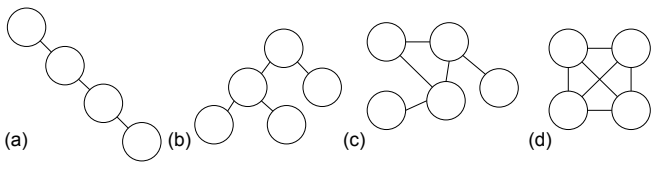


FIGURE 6.3: Pipeline, hierarchy, random structure, and fully connected structure

To summarise: *line*, *bus*, *ring* and *point-to-point* networks are essentially the same; and *tree* and *star* networks are also considered to be one topology. Therefore, from this point forward, we only refer to *fully connected*, *mesh*, *line*, and *tree* topologies. Indeed, we no longer refer to hardware networks, and instead exclusively discuss virtual networks. To highlight this distinction, we no longer refer to each of these as *fully connected*, *mesh*, *line*, and *tree* topologies, but instead refer to them as *fully connected structures*, *random structures*, *pipelines*, and *hierarchies* respectively.

6.3 Network Based Search

As discussed above, in decentralised service location where there is no central registry, agents must communicate with other agents in the system directly. As with the centralised approach, when locating services, an agent only communicates across the connections specified in the organisational structure, so the result of this approach depends on the structure used.

To help describe our service location approach, we assume the organisational structure shown by the graph in Figure 6.4, where the vertices k, n, m, p, q, and u are *agents*, and each edge is a *connection* between two agents. Here, rather than rely on the central registry to act as a matchmaker, agents can use their connections to other agents to locate a service. Additionally, if an agent's neighbours do not offer a required service, the agent can request that its neighbours, in turn, query their neighbours for the same required service. By iteratively performing this process, a single agent can search the entire network. For this reason, we refer to it as *network based search*.

To demonstrate, assume that agent k wants to locate a service s that is offered by agent m. There is no direct connection between k and m but, by using *network based search*, agent k can recruit its neighbours to locate s using their connections. Agent k first

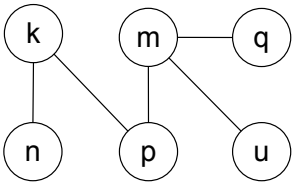


FIGURE 6.4: Example of some connected agents.

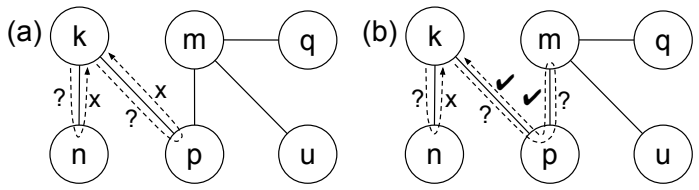


FIGURE 6.5: Network based search in action.

sequentially queries agents n and p whether either of them offer service s, as shown in Figure 6.5(a) where **?** represents a query. Both n and k reply stating they do not, represented by an **x**. Agent k then asks each of its neighbours to ask their neighbours, in turn, if they offer the required service, as shown in Figure 6.5(b). The first query is sent to agent n, but as n has no other neighbours, it immediately gives a negative reply. Next, k asks agent p about its neighbours. In turn, p asks its only other neighbour, m, for service s, and m confirms to p, who confirms to k, so that k can then allocate the task to m. This is shown in Figure 6.4(b), represented by a ✓.

By making use not only of its own connections, but also its neighbours' connections, an agent can thus perform a breadth-first search across the network to locate a service. While this approach in general provides an effective way of locating agents and their services, clearly its effectiveness depends heavily on the connections between agents. In particular, if the required service is not found until the search is many levels deep, then service location will take *significantly* more time, because the amount of communication required increases exponentially as *network based search* progresses.

There exist more complex approaches to reducing the number of messages needed to locate services, such as query routing [73], which uses forward knowledge (about the services provided by one agent) to reduce the amount of communication required when searching for a service provider. Though other approaches may reduce the amount of communication required, in general the more jumps between an agent and the service it requires, the more communication is needed to locate it, even with query routing. We have therefore chosen to adopt our more simple approach (*network based search*) and not use query routing, so that the added complexity does not distract from what we plan to investigate: can we reduce the distance between agents, and the service they require, thereby locating services faster?

6.4 The Effect of Structure on Network Based Search

The search path for *network based search* forms a tree structure, where the root vertex is the agent searching for a service, every other vertex in the tree is an agent that is queried for a particular service, each edge is a branch of the search path, and the children of each vertex represent agents being queried. For an agent to query a single neighbour, two time steps elapse: one to send the query; and another for the response. The time required in a given network can thus be determined by the number of connections in the search tree.

Suppose we have a search tree, as in Figure 6.6(a), such that below each vertex there are two branches: a binary tree. If we consider the root of the search tree to be level 0, then from level 0 to levels 1, 2, and 3 there are a total of 2, 6, and 14 edges in the search tree, respectively. Thus, for binary trees, from level zero to level or *depth*, d, there are $2^{d+1} - 2$ edges in the search tree. More generally, with any regular number of branches from each vertex, where the search tree is d levels deep, and each vertex has b branches, we have the following number of edges:

$$\frac{b^{d+1} - b}{b - 1}$$

As *network based search* progressively searches 1 level deep, 2 levels, and so on, until d levels, as in Figures 6.6(a), (b), and (c), respectively, the total time needed to reach d levels can thus be calculated as follows:

$$\sum_{y=1}^{d} \frac{b^{y+1} - b}{b - 1} \times 2$$

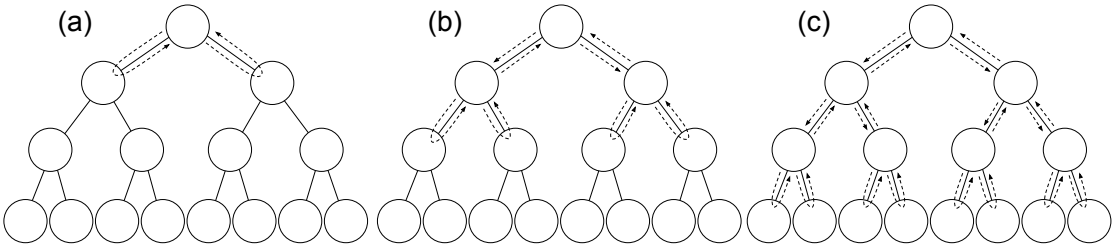


FIGURE 6.6: Network based search: search tree

where y is each value between 1 and d. This is fine when $b > 1$, but when $b = 1$ (essentially, when the search tree is a pipeline), this is not applicable, and the number of edges is simply the number of agents in the pipeline minus one, or the depth of the search tree: d. Thus, the time to search an entire tree to depth d, where each vertex in the search tree has b branches, is as follows.

$$searchTime(b, d) = \begin{cases} \sum_{y=1}^{d} \frac{b^{y+1} - b}{b - 1} \times 2, & \text{if } b > 1 \\ \sum_{y=1}^{d} 2y, & \text{if } b = 1 \end{cases} \tag{6.1}$$

If we know the depth of the search tree then, assuming the tree is *balanced* (each vertex has the same number of children), we can calculate the search time, which is shown to be exponential when $b > 1$ and polynomial when $b = 1$. While requiring a balanced tree is not, in general, appropriate, this does nevertheless provide an estimate. Clearly, the best case arises when the agent that initially receives a task can execute it itself, so that *service location time* is zero. In what follows, we estimate the *worst case* and *average case* for each topology separately (assuming a structure that consists of 100 agents).

6.4.1 Fully Connected Structure

In a *fully connected structure*, the search tree is never more than one level deep, so search is similar to the *central agent registry* approach where *sessions* are maintained. The only difference is that no communication is required between the agent and the registry, so services are located more quickly, causing any changes in performance to occur more rapidly.

For 100 agents, each agent is connected to 99 other agents, so the worst case for *service location time* using Equation 6.1, is as follows, where $d = 1$ and $b = 99$.

$$\sum_{y=1}^{1} \frac{99^{y+1} - 99}{99 - 1} \times 2 = \frac{99^2 - 99}{99 - 1} \times 2 = 198 \text{ time steps}$$

Because the search never traverses more than one hop away from the searching agent, when only one instance of a service exists *service location time* increases linearly as more agents are searched. This meaning that the average case will be half of the worst case,

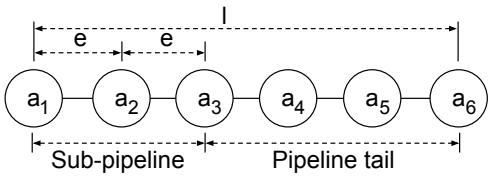


FIGURE 6.7: Determining the worst case for service location time for agent a_2

which is 99 time steps. The search tree is regular regardless of the agent performing the search, with all agents having exactly 99 connections, and the search tree always 1 level deep.

6.4.2 Pipeline

In a *pipeline*, the worst case arises when agents a_1 and a_2 are at opposite ends of the structure, and agent a_1 requires a service only offered by agent a_2. So, for a set of 100 agents in a pipeline, the search tree for an agent at the end of the pipeline has a maximum depth of $d = 99$, and each vertex only has $b = 1$ branch. Therefore, using Equation 6.1 gives the worst case as follows.

$$\sum_{y=1}^{99} 2y = (1 + 2 + \cdots + 99) \times 2 = 9900 \text{ time steps}$$

However, if an agent a is in the middle of the pipeline, with 50 agents to one side, and 49 agents to the other, then a essentially has a pipeline on either side. If we assume, for simplicity, that both pipelines have a length of 50, so each has a search of depth $d = 50$, the worst case for *service location time* is the time to search two pipelines of length 50 agents. We can determine the worst case by using Equation 6.1, and multiplying the result by two, as follows.

$$\left(\sum_{y=1}^{50} 2y \right) \times 2 = \sum_{y=1}^{50} 4y = 5100 \text{ time steps}$$

Now, for any other agent, this is more complicated because the number of agents at each side is different, so we divide the pipeline as in Figure 6.7, where we focus on agent a_2. We require e, the distance to the nearest end of the pipeline, and l, the length of

the pipeline. Then, to calculate the worst case, we divide the problem into three parts. First, we calculate the time to search the portion of the pipeline, or the sub-pipeline, in which the searching agent, a_2, is the middle agent. The sub-pipeline starts at a_1, ends at a_3, and is of length $2e$. As stated previously, if an agent is in the middle of the pipeline we can determine the search time by calculating the time to search the pipeline on one side using Equation 6.1, then multiplying the result by two, as follows.

$$\begin{aligned} &\sum_{y=1}^{e} 2y \times 2 \\ =&\sum_{y=1}^{e} 4y \end{aligned} \tag{6.2}$$

Second, we determine the time to search the remaining portion of the pipeline. Because we have searched the sub-pipeline of length $2e$, the remaining portion of the pipeline, or the *pipeline tail*, is of length $l - 2e$. We can determine the time to search it using Equation 6.1, as follows.

$$\sum_{y=1}^{l-2e} 2y \tag{6.3}$$

Finally, whenever an agent in the *pipeline tail* is searched, a_2 must also communicate through the entire length of the *sub-pipeline*; in our example, this is a_1 and a_3. Therefore, we also calculate the time needed for agent a_2 to search to the outermost agents in the sub-pipeline ($2e \times 2$), and multiply it by the number of agents in the pipeline tail ($l - 2e$), as follows.

$$\begin{aligned} &(2e \times 2) \times (l - 2e) \\ =&4e(l - 2e) \end{aligned} \tag{6.4}$$

By adding together the result of equations 6.2, 6.3, and 6.4, for an agent that is e edges away from the closest end of the pipeline, in a pipeline that has a length l, the worst case for *service location time* can be calculated as follows.

$$\left(\sum_{y=1}^{e} 4y\right) + \left(\sum_{y=1}^{l-2e} 2y\right) + (4e(l - 2e)) \tag{6.5}$$

The worst case for *service location time* for a pipeline of 100 agents, for each agent in order, is shown in Figure 6.8, in which it is clear that from the centre, radiating

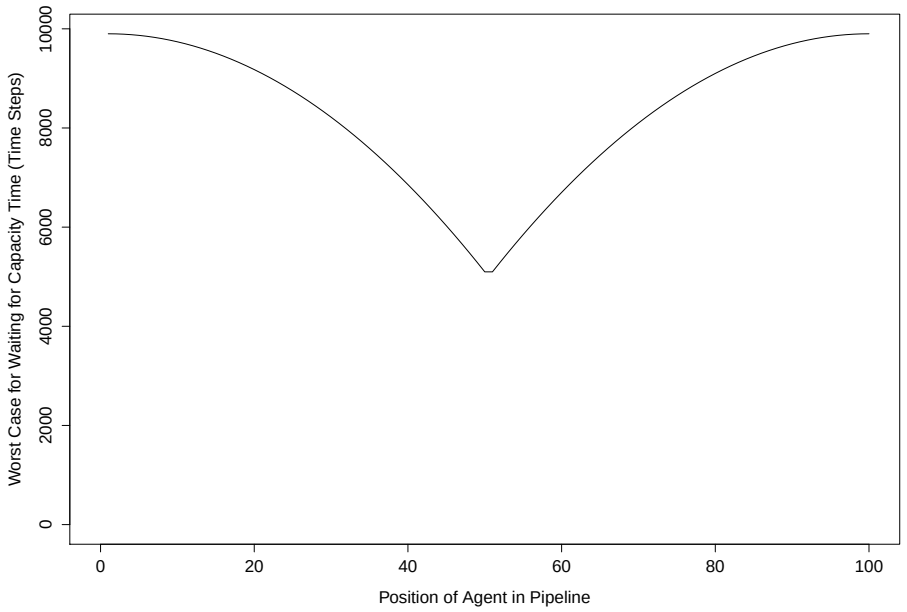


FIGURE 6.8: Worst case for service location time in a pipeline, depending on position

outwards, the worst case initially increases rapidly, and then slows down, but continues increasing. For a pipeline of 100 agents, the worst case is 9900 time steps, but for the two centre agents the worst case is 5098 time steps.

Average *service location time* also varies according to the position of the agent, but since the lowest is achieved by an agent in the middle, and the highest by an agent at the end, a range can be determined within which lies the average for the pipeline as a whole. Now, on average, half of all agents need to be searched, so for a pipeline containing 100 agents, the average *service location time* for the middle agent is as follows, where $d = 50/2$.

$$\sum_{y=1}^{25} 4y = 1300 \text{ time steps}$$

For an agent at the end of the pipeline, the average time is as follows, where $d = 50$.

$$\sum_{y=1}^{50} 2y = 2550 \text{ time steps}$$

Thus, the average *service location time* for a pipeline of 100 agents is between 1300 and 2550 time steps. It is interesting to note that if a pipeline is in a ring configuration then

every agent is in the middle of the pipeline, and so the average and worst case for service location time will be the same, no matter where the agent is in the pipeline.

In summary, the search tree for pipelines is not regular. Nevertheless, it is clear that even if an agent is in the middle of the pipeline (the position that achieves the best performance), the worst case and average case for *service location time* are much worse than that achieved in a *fully connected structure*. The worst case is $5098/198 \approx 25$ times longer, and the average case is $1300/99 \approx 13$ times longer.

6.4.3 Hierarchies and Random Structures

Though we have been able to determine the worst case and average case for *service location time* for *pipelines* and *fully connected structures*, performing a similar analysis for a *hierarchy* and a *random structure* is difficult. This is because the connections in each can have a large number of configurations. For a hierarchy the number of overall connections is consistently the number of agents in the structure, minus one, but a single agent can have a large number of connections or very few. For a *random structure* the number of overall connections in the structure can also vary.

To determine the time to search an entire search tree, we require two parameters: the number of children below each agent in the tree, b; and the depth of the tree, d, as shown in Equation 6.1. Given that these values change from structure to structure, and even within a single structure these values are not uniform, we cannot provide an accurate determination for *service location time*. However, by providing an average value for b, and an average and maximum value for d, we can *estimate* the worst case and average case for service location time. For example, if the search tree has an average of $b = 2$ children at each level, and if the furthest distance between two agents in the network structure being searched is $d = 7$ and the average distance is $d = 5$, then using Equation 6.1, we could estimate the worst and average case for *service location time* as follows.

$$\sum_{y=1}^{7} \frac{2^{y+1} - 2}{2 - 1} \times 2 = 988 \text{ time steps (worst case)}$$

$$\sum_{y=1}^{5} \frac{2^{y+1} - 2}{2 - 1} \times 2 = 228 \text{ time steps (average case)}$$

However, it is important to note that this equation can be used only to derive an *estimate*, not an accurate result, because an actual search tree would have a variable depth for different branches, and the number of children would also very throughout the tree.

6.5 Experimentation and Results

To evaluate the performance of *network based search*, and examine the effect of structure on the performance of decentralised service location, we performed four experiments (one for each structure), using the simulated framework described in Chapter 4, and the experimental parameters described in Section 4.4. However, since decentralised service location can be applied to the variety of structures described above, we must specify how each of these structures is generated so that they can be used when simulating *task allocation and execution*.

Creating a *fully connected* structure is trivial, because there is only one configuration of connections for such a structure (every agent is connected to every other agent). Provided that we assume that entities in a *pipeline* structure are ordered arbitrarily, generating a *pipeline* is also trivial. Constructing a *hierarchy* is more complicated, but we have already specified how to generate a *hierarchy* in Section 4.2.2 when generating a *tree* of ordering constraints for a task, and we use the same approach here. Therefore, we need only describe how to create a *random* structure. There exist many models for generating random graphs. For the purpose of modelling social networks with scale free networks, Albert and Barabási's model is suitable [6]. To generate a graph that exhibits small world properties, we would use Watts and Strogatz's model [121]. However as our intention is simply to provide some random connections, in contrast to the regular connections found in a pipeline or hierarchy, Erdős and Renýi's model is sufficiently complex for our pruposes [35].

In this model, a graph $G(\mathrm{N}, p)$ has all pairs of nodes in N connected with probability p. Ideally, this structure consists of as few connections as possible, to distinguish from a fully connected structure. However, as we have already discussed, we want to ensure that the structure is *connected* so that a path exists between all pairs of agents. To achieve this we rely on Erdős and Renýi's variation of this graph, as follows:

$$G = \left(\mathrm{N}, \frac{2 \ln |\mathrm{N}|}{|\mathrm{N}|}\right)$$

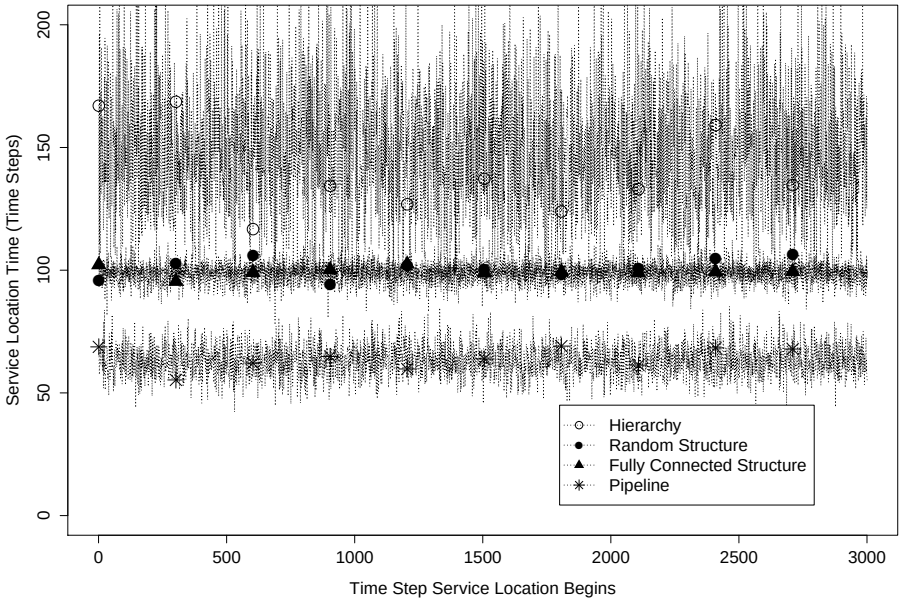


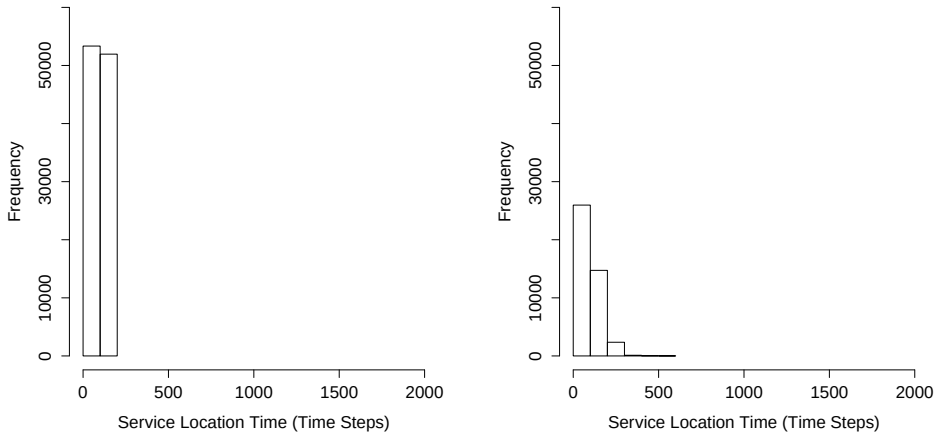
FIGURE 6.9: Decentralised service location: service location time comparison

This variation uses the number of vertices, $|N|$, in the graph to calculate the lowest value of p such that a connected graph is produced, as illustrated in Figure 6.3(c).

When using a central registry to locate services — with the exception of the *isolated requests* — each method of service location has a well defined upper limit for *service location time*, as discussed in Chapter 5. However, as the distance increases between an agent and the service it is trying to locate, the number of messages increases exponentially. Because the number of messages can increase rapidly, and many agents are locating services simultaneously, the computational complexity of simulating this process forced us to impose a limit on the *network based search* as follows. The *network based search* essentially performs an iterative deepening depth-first search across the network of agents. We impose a limit on the depth of the *network based search*, such that if the service has not been located after searching 20 connections away from the searching agent, then the search fails, as does the task.

6.5.1 Fully Connected Structure

As discussed above, if *network based search* in a system of 100 agents operates over a *fully connected structure*, then *service location time* will take, on average, 99 time steps,



(a) Fully Connected (b) Random Structure

FIGURE 6.10: Decentralised service location: service location time frequency distribution

and a maximum of 198 time steps. This is supported (as expected) by the results from our first experiment, shown in Figure 6.9, in which average *service location time* is indeed approximately 99 time steps (midway between the minimum and maximum service location time). By examining the frequency of *service location time* in Figure 6.10(a) we can also see that service time is consistently between 0 and 198 time steps, and the frequency of service location time is evenly distributed across this range (with approximately half of the instances between 0 and 100, and half between 101 and 200).

Recall that the key difference here with centralised approaches is that fewer messages are needed to achieve similar results. Thus, the number of overloaded agents, average load and waiting for capacity time are all similar to the centralised approaches. Figure 6.11 shows experimental results confirming that the number of overloaded agents remains relatively small, and is never sustained above zero. Here, each point in the graph indicates the number of overloaded agents at a single point in time (time step in the simulation). Similarly, Figure 6.12 shows that the average load increases, like the centralised approaches that use *isolated requests* and maintain *sessions* (in Figure 5.4), but at a faster rate. Both centralised and decentralised approaches reach a plateau of 50% load, but here it is much faster (after approximately 500 rounds, compared to over 1000 rounds for *sessions*, and 2500 rounds for *isolated requests*). Finally, Figure 6.13 shows that for *waiting for capacity time*, the plateau is reached more rapidly than in the

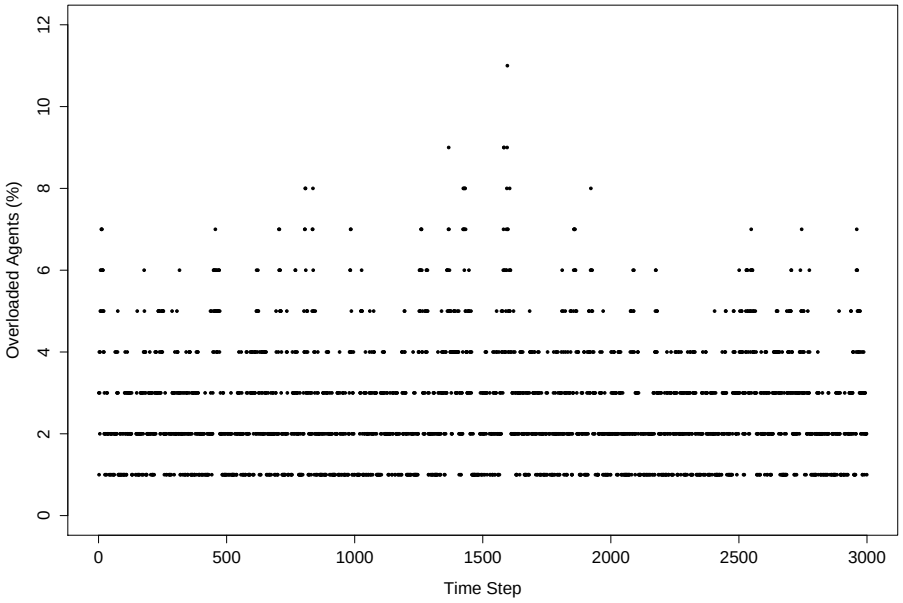


FIGURE 6.11: Fully connected structure: number of overloaded agents

centralised approaches because services are located more quickly, and therefore tasks are allocated more rapidly.

Despite this positive aspect of *network based search* on a *fully connected structure*, the use of such a structure is not feasible for realistic scenarios in which the devices or services are large in number. It is not reasonable to assume that all agents can maintain an internal registry of all others. Instead, this simply provides a theoretical baseline to compare against, without the cost of each agent maintaining total knowledge.

6.5.2 Random Structure

For random structures, we undertook an experiment with 100 agents, generating graphs as specified in Section 6.5, which have on average 9.2 connections per agent, compared to exactly 99 connections for each agent in the fully connected structure. Now, with fully-connected structures, any service is only one connection away from a service consumer since each agent is connected to all others. With centralised approaches, a service is two steps away from a consumer, since it requires only a request to the registry and a request from the registry to the provider. In contrast, with random structures, this distance is limited only by the number of agents, with that number being the maximum number

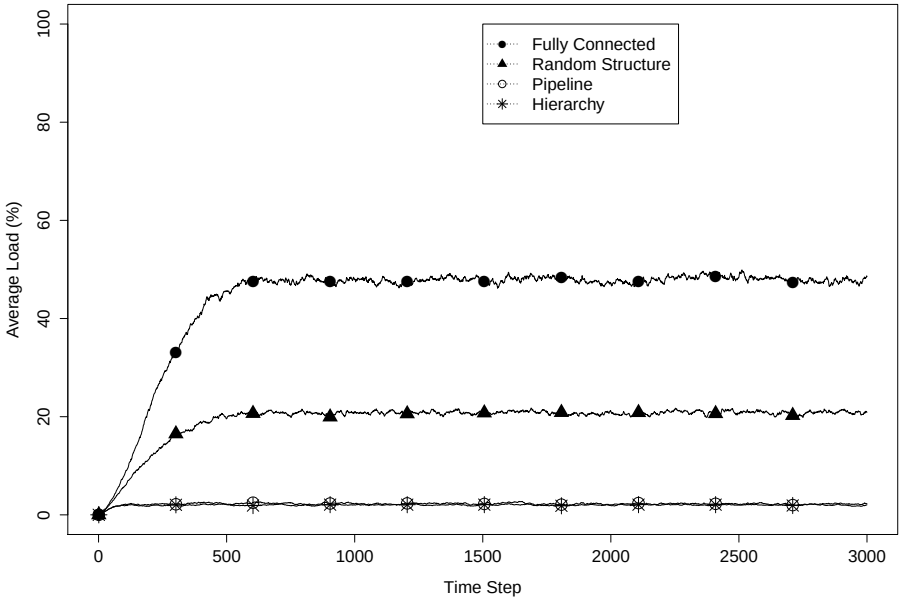


FIGURE 6.12: Decentralised service location: average Load

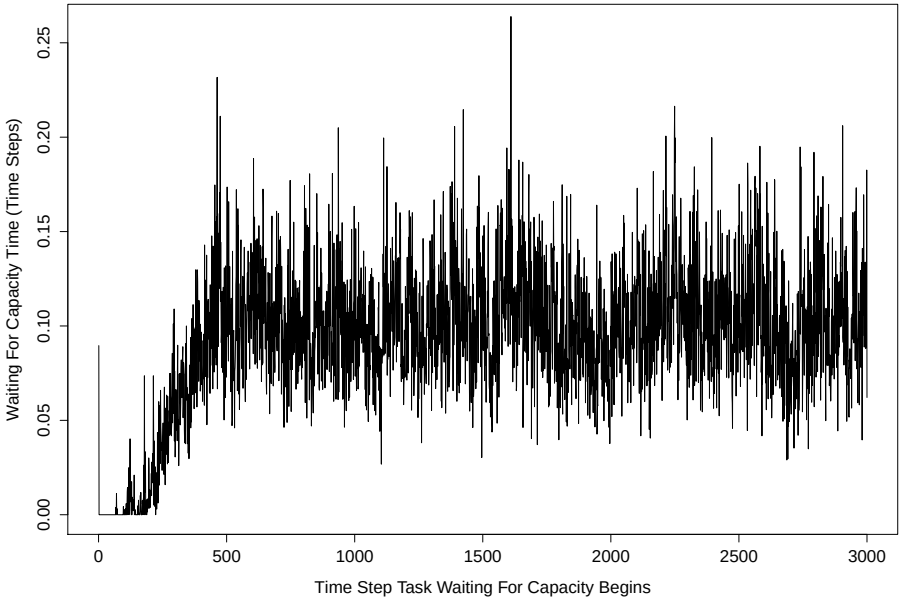


FIGURE 6.13: Fully connected structure: average waiting for capacity time

of hops. However, as discussed above, we limit the search through such structures to a maximum of 20 hops in order to make the simulations manageable; this brings some consequences for our results. In particular, it may not be possible to locate some services within this limit of 20, so service location, and the originating task, may therefore fail.

In Figure 6.9 we can see that the average time to locate a service is around 100 time steps. Figure 6.10(b) also shows that the service location time frequency distribution is smaller than the *central agent registry* with *isolated requests*, and service location takes longer than a *central agent registry* maintaining *sessions* on only very rare occasions (as can be seen by comparison with Figure 5.2(b)).

On first sight, the results seem to show that using the *network based search* on a *random structure* performs just as well as when used on a *fully connected structure*, with a 90% reduction in the number of connections that need to be maintained. However, on closer inspection, we find that approximately half of all required services are not located. This is evident when comparing the frequency distribution of *random structures* to that of *fully connected structures*. The frequency distribution of *service location time* also shows the total number of services located, and in Figure 6.10(a) we can see that, across a *fully connected structure*, just over 100,000 services are located, whereas in Figure 6.10(a) we can see that, across a *random structure*, approximately 45,000 services are located — under half.

We know that the set of 45,000 service location processes are represented in our results. We also know that 55,000 are not represented because the service location processes either failed, or are still running when the experiment finishes. If a service is located quickly, then it is used in our results, and if the service location process fails, or does not finish, then it is omitted, therefore it is safe to assume that average *service location time* will, in fact, be much higher than our results show.

From our analysis of *network based search* in Section 6.4 we argued that we can estimate the time to locate services based on the average and maximum distance between all agent pairs, and the average number of connections for each agent. In the random structures that we generate, each agent has an average of $d = 9.2$ connections, the average distance between all agent pairs is between $d = 2$ and $d = 3$, and the worst case is $d = 4$. Using Equation 6.1, we can calculate that average *service location time* is between 198 time steps (for $d = 2$), and 1836 time steps (for $d = 3$). We can also calculate that the worst case for *service location time* is approximately 16,956 time steps (for $d = 4$).

We can therefore conclude that when our simulation completes, a large number of services are still being located including, potentially, some search processes that started at time step 1. More broadly, we can also conclude that reducing the number of connections will increase service location time, and so increase the number of services that are not located within the duration of our experiments.

As a large proportion of service location processes do not finish, few tasks are being successfully allocated to increase the system's load, and so it is reasonable to assume that this will cause the average load to be relatively low compared to centralised and fully connected systems. As the system's load will be lower, the number of overloaded agents will also be lower, causing *waiting for capacity time* also to be lower.

Figure 6.14 shows that this is, indeed, the case as agents are almost never overloaded. In addition, Figure 6.15 shows that tasks, on average, spend less time waiting to be executed. Given that approximately half of all tasks are not successfully allocated to agents that can execute them, it stands to reason that the number of overloaded agents, and the average time waiting for capacity is going to be lower. The true detrimental effect of this approach can be seen in Figure 6.12, which shows that the average load of all agents does not breach 20%. Though we could hope that this is because the approach makes the system significantly more efficient, executing all tasks with significantly less load, this is not the case.

6.5.3 Pipelines and Hierarchies

To examine pipelines and hierarchies, we undertook two further experiments in the same way, and to a limited depth. This limitation causes task failure in many cases, and so our results show little. However, for pipelines, the fact that the majority of tasks fail supports our analysis in Section 6.4.2, which shows that *service location time* is extremely poor compared to *fully connected structures* and centralised systems.

More generally, because the amount of time required to locate a service increases polynomially in a pipeline, *service location time* is poor, unless services are very close to the agents that require them. In consequence, the majority of services cannot be located, so most tasks cannot be allocated, and the average agent load is extremely low (Figure 6.12). We also cannot adequately assess *waiting for capacity time*.

For hierarchies, Figure 6.9 shows that average *service location time* is poor, at around 150 time steps. There are few data points because, as with *pipelines*, most service

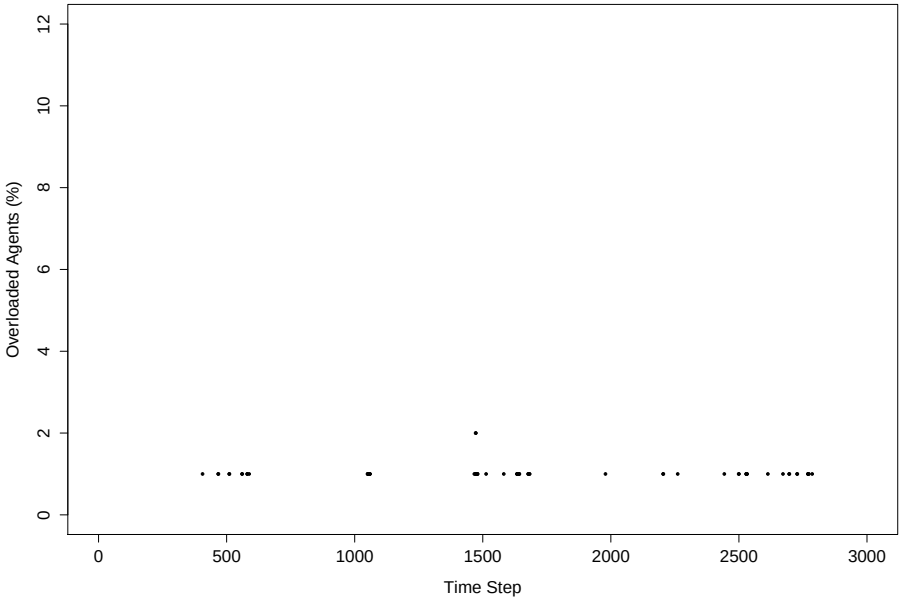


FIGURE 6.14: Random structure: number of overloaded agents

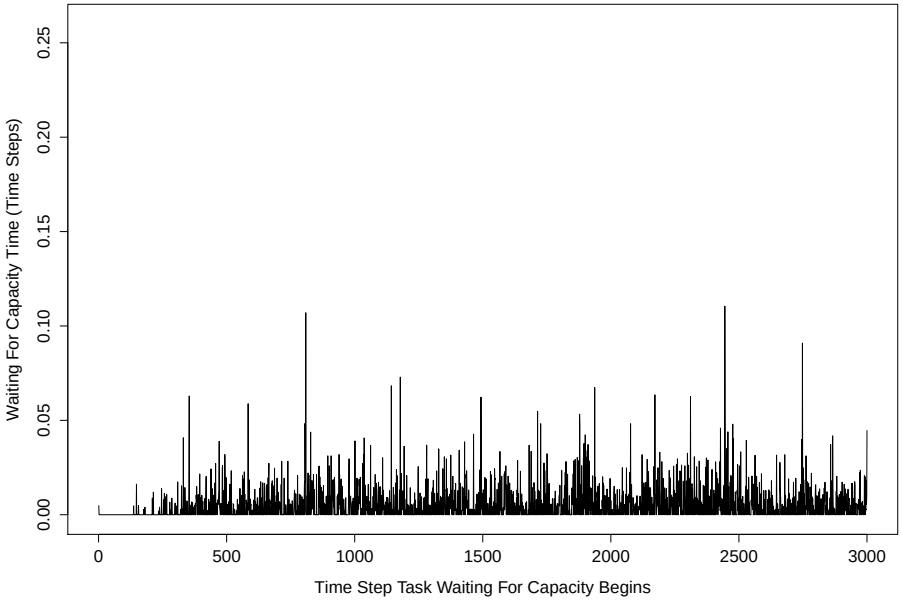


FIGURE 6.15: Random structure: average waiting for capacity time

location processes fail or do not finish, though this is better than for *pipelines*. Similarly, Figure 6.12 shows that average agent load is extremely low, but this is caused by poor service location performance.

In the same fashion, as with *random structures*, we can determine the average number of connections for each agent ($d = 3$), the average distance between each agent pair (which is between $d = 5$ and $d = 6$), and the maximum distance between all agent pairs ($d = 7$). Using Equation 6.1, we can calculate that average *service location time* is between 1,074 time steps (where $d = 5$) and 3,258 time steps (where $d = 6$), and the worst case is approximately 9816 time steps (where $d = 7$).

6.5.4 Discussion

The results above suggest that there is a trade-off between the total number of connections in a structure, or the structure's level of *connectivity*, and *service location time*. More specifically, when a set of 100 agents is connected by a *fully connected structure*, each agent has exactly 99 connections, with a total of 4950 connections in the structure. Alternatively, when the same set of agents is connected by a *random structure*, each agent has an average of 9.2 connections, with approximately 455 total. In our experiments, a *hierarchy* always has a total of 99 connections, and the same is true for pipelines.

We can see that as the level of *connectivity* decreases, *service location time* increases. Among all these structures, a *fully connected structure* achieves the lowest *service location time*, but also has the highest level of *connectivity*. In contrast, a *pipeline* has the lowest level of *connectivity*, but results in the highest *service location time*. A *random structure* lies somewhere in between for both *connectivity* and *service location time*.

However, drilling down into the detail, there is also a more subtle point to notice here. Both the *pipeline structure* and *hierarchy* have the same number of connections, but the *hierarchy* performs better than the *pipeline* (though only marginally). In addition, the centralised service location approaches in Chapter 5 had the same number of connections as both the *pipeline* and the *hierarchy* yet they achieved much lower service location time. This is because an increase in *connectivity* is indicative of a decrease in service location time, but the actual reason *service location time* decreases is because the average *distance* between each agent, and all services, decreases. By decreasing the distance between agents and the services they require, services can be located quicker.

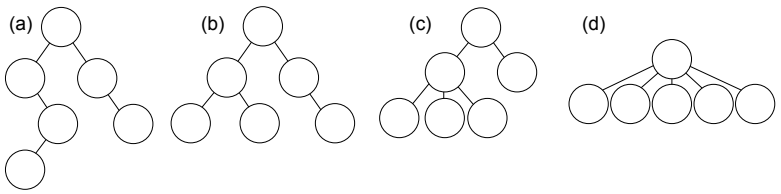


FIGURE 6.16: Degree frequency distribution in various structures

Though the average distance between services and agents can be decreased by increasing the overall number of connections in the structure, we can also produce the same result by changing the existing connections. For *pipelines* and *hierarchies*, the number of connections is the same, but the frequency distribution of the degree of connectivity of all agents is different. This can be seen in Figure 6.16, which contains four hierarchical structures. In each structure, the number of agents, and the number of connections, are the same, but each agent's *degree of connectivity* varies considerably. In each structure the minimum degree of connectivity is always 1, but from 6.16(a) to (d) the maximum degree is 2, 3, 4, and 5, respectively. At the same time we can observe that the maximum distance between any two agents also decreases.

As a result, as *service location time* can be improved by significantly increasing either a structure's level of *connectivity*, or its *degree distribution*, but doing either is not ideal. Increasing a system's *degree distribution* results in bottlenecks, where agents with high degree receive a disproportionately large number of queries during the *service location process*, while increasing a structure's level of *connectivity* is not always sensible (or indeed feasible), as discussed when considering *fully connected structures*.

Figure 6.17 illustrates the interplay between *service location time*, the level of connectivity, and *degree distribution*. The lowest *degree distribution* is produced when all agents have the same degree of connectivity, and the highest when one agent is connected to all other 99 agents (with all others having only one connection). The lowest level of *connectivity* is 99 connections, in a *pipeline*, *hierarchy*, or centralised system, and the highest level of connectivity is a *fully connected structure*, resulting in 4950 connections. Finally, the lowest average *service location time* achieved in our results is two time steps, and as our simulations lasted 5000 time steps, and some services are not located in that time, we can say conclusively that the highest *service location time* is greater than 5000 time steps.

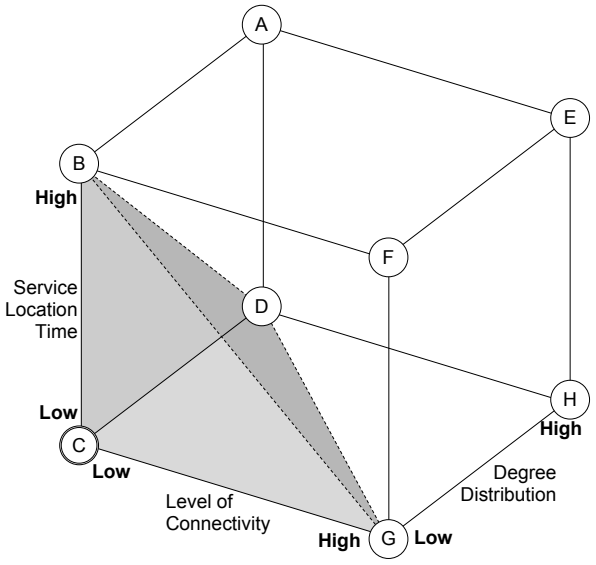


FIGURE 6.17: The effect of degree frequency distribution, and the total number of connections, on service location time

An ideal system is situated at point C, such that the structure's level of *connectivity* is low, the degree distribution is the same for all agents so that no bottlenecks form, while services can be located rapidly. However through the experiments performed in this chapter and Chapter 5, we have found that this is difficult to achieve.

For example, consider all three centralised approaches, which have fixed structure. They each have the highest possible *degree distribution* because the *central registry* has a degree of connectivity of 100, while all other agents have a degree of 1. They also have the lowest level of *connectivity* of the number of agents minus one. Therefore, in Figure 6.17, all centralised approaches appear somewhere on the line between A and D. When *isolated requests* are used, average *service location time* is 400 time steps, appearing about half way between A and D. With *sessions*, average *service location time* is 200 time steps, and so is closer to D. Finally, using a *central service registry* achieves the best possible *service location time* and so is situated at D.

Each of the four decentralised approaches is much more varied. Average *service location time* for a *fully connected structure* is approximately 100 time steps, with extremely low *degree distribution* because every agent has the same degree of connectivity (99 connections). However, this structure also has the highest possible level of *connectivity*, so the result must be placed between point F and G in Figure 6.17. For a *random*

structure the level of *connectivity* is much lower but *degree distribution* now forms a bell curve (so is moderate), and *service location time* is higher than for *fully connected*. This can therefore be placed directly in the middle of the cube, because it performs moderately across all characteristics. Finally, both *pipelines* and *hierarchies* have the lowest possible level of *connectivity*, and low degree distribution (but slightly higher for hierarchies). However, for both structures, performance is extremely bad, with the highest results for average *service location time*. Therefore, a *pipeline* is situated at B, while a *hierarchy* is close to B, but tends towards A and C.

As stated earlier, our aim is to achieve the lowest *service location time* possible, while decreasing the structure's level of *connectivity*, and *degree distribution*. However, as we have just shown, by lowering one of these levels, the other two tend to be increased.

6.6 Conclusion

This chapter has demonstrated how to remove a system's reliance on a *central registry* by introducing a decentralised approach to locate services, which we call *network based search*. We then evaluated the performance of this approach when operating on top of a *fully connected structure*, *random structure*, *pipeline*, and a *hierarchy*.

To summarise, to improve overall system performance, we need to design a system such that the structure's level of *connectivity*, and its *degree distribution*, are as low as possible, while achieving low *service location time*. However, designing a system that achieves these three characteristics is difficult because decreasing one of these three factors often increases the other two. In fact, in Figure 6.17 the grey pyramid between points B, D, G, and C, shows the area that no approach manages to enter. In the next chapter, we explore two alternative approaches to decrease *service location time*, while maintaining low *degree distribution* and *connectivity*.

Chapter 7

Improving Task Throughput via Structural Adaptation

7.1 Introduction

It is clear from the review in Chapter 2, and from Chapter 6, that because the connections between agents guide the *service location process*, choosing the right organisational structure is just as important to the system's overall performance as the agents themselves. If this structure is inappropriate for the task faced by agents, then their efficiency, in particular in locating agents to which to delegate tasks, is compromised, since it can take a long time to locate needed services. Moreover, if the environment is dynamic, as might be expected in any real-world environment of the kind we are considering, where openness is key characteristic, with agents joining and leaving the system at will, changing the services they offer and, importantly, changing their network connections, then such problems can become even more significant. Since the network structure is fundamental to efficiency, a change to this structure should be able to improve (or decrease) it. Such adaptation is the focus of this chapter.

In any consideration of adaptation, there are two key issues to address. First, we need to examine what adaptations can be introduced and how the network structure changes as a result. Second, we need to examine the effects or consequences of these adaptations, in order to ensure that they are beneficial: that they improve overall system performance or efficiency. Now, in all this we are guided by the principle of Occam's

Razor, of parsimony, economy, or succinctness, and seek not to provide overly elaborate algorithms for adaptation when simpler ones will do. In this context, our concern is not to complicate adaptation, but instead to provide an analysis that reveals where and how a parsimonious approach can reap benefit.

Thus, in contrast to other efforts, which have sought merely to adopt overall metrics that provide a global evaluation of performance, through some measure of utility, for example, in this work we seek, first, to provide simple approaches to reorganisation and, second, to evaluate these approaches in the context of the explicated process of task allocation and execution.

This chapter is structured as follows. In Section 7.2 we propose two structural adaptation techniques for reducing the distance between agents and the services that they require, to improve overall performance when locating services. Section 7.3 discusses some potentially dangerous behaviour from each structural adaptation approach, and some techniques to avoid them. Our reorganisation approach is then used in Section 7.4, and each is evaluated. Finally, in Section 7.5 our evaluation technique is compared to some related work, and a potential application for our approaches is discussed. The chapter is concluded in Section 7.6.

7.2 Structural Adaptation

As we have seen in Chapter 6, *service location time* is affected by the distance between agents, and the services they require since, if the distance from an agent to the service that it requires is small, then the service can be located quickly. In response, in this section, we introduce two approaches for structural adaptation that aim to take advantage of this relationship between *service location time* and the distance between agents. We can consider these approaches to be *distance reduction reorganisation* approaches, the aim of which is to reduce the time required to locate services by decreasing the number of neighbours that need to be contacted when using *network based search*. The key issue for these approaches to address is to ensure that the direct connections between agents are to those offering required services.

Based on the assumption that if a service has been used in the past it is likely to be used again, the most simple means of reducing distance is to adapt the organisational structure so that when an agent encounters a new agent that offers a required service

it creates a direct connection to it for subsequent use. We can also consider a more sophisticated approach in which agents have connections to those offering the most *frequently* used services. Given our premise of *requirement correlation*, the assumption that there is a correlation between services that any particular agent will need, it may be possible to identify those agents (and services) that can satisfy the relevant needs. Thus, a second approach seeks to ensure that each agent's most frequently used services are offered by at least one of its neighbours. Both of these approaches require additional information such as the services an agent has encountered, and the frequency in which services are used. Therefore, each of these approaches also relies on adapted forms of *network based search*, such that this information is recorded.

However, before considering either of these approaches, it is important to determine whether adequate adaptation can also be provided without leveraging this extra information about services and their use, both in itself, and as a baseline for comparison. We therefore begin by introducing an approach that simply offers random changes to the organisational structure.

7.2.1 Random Reorganisation

To distinguish arbitrary structural changes from those based on prior knowledge of service use, we introduce a *structural adaptation* approach that we call *random reorganisation*, which simply changes random connections at each time step. This is achieved by locating a number of random agent pairs that are connected, removing the connection between them, locating a number of random agent pairs that are not currently connected, and creating a connection between each of these pairs. This approach is formally specified in Algorithm 2, where A is the set of all agents, and nc is the number of connections to be added and removed. The algorithm makes use of a function, $rand(\text{A})$, which returns a random agent from the set of agents A, a function $connectedTo(a)$, which returns a set of agents to which agent a is connected, and two functions $createConnection(a_1, a_2)$ and $removeConnection(a_1, a_2)$ which, respectively, create and remove a connection between a_1 and a_2.

7.2.2 Full Service Connection Reorganisation

As we have stated, our aim here is for *distance reduction reorganisation* on the premise that if an agent is directly connected to at least one agent offering each of the services

Algorithm 2 randomReorganisation(A)

```
1. for i = 1 to nc do
2.     a1 ← rand(A)
3.     a2 ← rand(connectedTo(a1))
4.     removeConnection(a1, a2)
5.     a1 ← rand(A)
6.     a2 ← rand(A)
7.     while a2 ∈ connectedTo(a1) do
8.         a2 ← rand(A)
9.     createConnection(a1, a2)
```

that it requires, then it can always locate required services quickly. To enable agents to adapt connections according to the services that have been required, we need an adapted form of *network based search*, such that every time a service is required, it is added to the set of required services, RS. In *full service connection* reorganisation, when an agent requires a service that is not offered by an agent to which it is directly connected, it creates a connection to another agent that offers the service, located using our adapted from of *network based search*.

This is specified in Algorithm 3, where a_1 is the agent adapting its connections, RS is the set of services that have previously been required for tasks by agent a, and ES is the set of known (which we call *encountered*) services offered by agents to which a_1 is directly connected. Two functions are used here: *connectedTo*(a), which returns the set of agents to which agent a is connected; and *offeredBy*(a), which is the set of services of which agent a offers an instance. Lines 1–3 state that for every service that has previously been required for a task (and located), if that service is not yet offered by an agent to which a_1 is connected, then the service is added to the set ES, otherwise the next service in RS is considered. Next, lines 4–6 states that, for all agents to which a_1 is connected, if any offer the newly encountered service, then the flag *found* is set to *true*. Finally, on line 7, if at least one neighbour offers service s (that is, if *found* is *true*), then we move on to the next service $s \in$ RS. If no neighbours offer service s (that is, if *found* is *false*), then we continue to lines 8–9, where we locate an agent a_2 that offers service s using *network based search*, and create a connection between agents a_1 an a_2.

Algorithm 3 fullServiceConnection(a_1, RS, ES)

```
1. for s ∈ RS do
2.   if s ∉ ES then
3.     ES ← ES ∪ {s}
4.     found ← false
5.     for n ∈ connectedTo(a_1) do
6.       found ← found ∨ [s ∈ offeredBy(n)]
7.     if ¬found then
8.       a_2 ← networkBasedSearch(a_1, s)
9.       createConnection(a_1, a_2)
```

7.2.3 Frequent Service Connection Reorganisation

Now, as discussed in Chapter 3, we adopt the assumption of *requirement correlation*, by which some services are typically needed in combination, while others are not. Essentially, if one service has previously been required for a task, then any further services required by the task will be from a subset of services, or a *service category*. *Frequent Service Connection Reorganisation* is the second distance reduction reorganisation considered, and builds on this by ensuring that for each agent, at least one of its neighbours offers each of its most *frequently* used services.

To achieve this, we first require an adapted form of *network based search*, which maintains a counter for the number of times that each service has been required, and every time a service is required its counter is incremented. *Frequent service connection* can then make use of the function *sortByFrequency*(S), which sorts the set of services, S, according to the frequency that services have been required in the past.

Frequent service connection periodically checks that the most frequently used services are offered by one of an agent's neighbours. If a neighbour does not offer the service, then a connection is created to another agent that does offer it, located using *network based search*. This is specified in Algorithm 4, where a_1 is the agent adapting its connections and RS is the set of services that agent a_1 has previously required (as for full service connection).

All previously required services, RS, are first sorted by the function *sortByFrequency*(RS) (line 1), and the most frequently used services are placed in the set F (line 2). The flag, *found*, is set to *true* if there is at least one agent to which a_1 is connected that offers service s for all such services in the set of most frequently used services F (lines 3–6). Finally, on line 7, if at least one neighbour offers s (that is, if *found* is equal to *true*),

Algorithm 4 frequentServiceConnection(a_1, RS)

```
1. sortByFrequency(RS)
2. F ← top(RS)
3. for s ∈ F do
4.    found ← false
5.    for n ∈ connectedTo(a) do
6.       found ← found ∨ [s ∈ offeredBy(n)]
7.    if ¬found then
8.       a2 ← networkBasedSearch(a1, s)
9.       createConnection(a1, a2)
```

then we consider the next service in F. If no neighbour offers service s (that is, *found* is equal to *false*), then in lines 8–9 an agent, a_2, that offers s is found using *network based search*, and a connection is created between a_1 and a_2.

7.3 Network Disconnection and Failing Tasks

The *structural adaptation* approaches described above make changes to the organisational structure without any consideration of whether there is any consequence of the change. While the latter two use prior knowledge of service use in order to select which connections to break and make, they are otherwise unconstrained. Allowing agents to change connections without constraint in this way also allows agents to make any improvement they need. However, it also does not seek to ensure that the organisational structure remains connected, and one potential consequence is that the network of connections that allows agents to communicate with others, either directly or indirectly, can be partitioned into a number of separate networks. This is problematic; as discussed in Chapter 6, if the connections that link agents do not form a *connected* structure, then it is potentially impossible to locate some services. Since we have not previously been concerned with dynamic structures, such a consideration was only relevant at design time in the initial design of the organisational structure. However, in the context of introducing dynamic structures in which connections change during the operation of the system, this can be a very serious problem.

Figure 7.1 illustrates this danger. If agent k, during the structural adaptation process, removes its connection to agent m, then m can no longer reach agents k, p, q, and n. When agent m receives a task that requires a service that only agents q and n offer, it will use *network based search* to try to locate them. If the service location process is

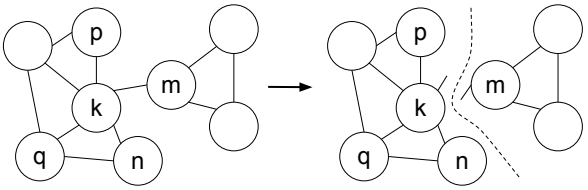


FIGURE 7.1: A connection removal causing graph disconnection.

performed before the connection between m and k is removed, then both q and n can be reached through k. However, after the connection has been removed, q and n are unreachable. To ensure that disconnection does not affect service location, we can either take measures to prevent the network from disconnecting, or take measures to reconnect the network when the network is disconnected. We call these *preventative* and *remedial* approaches, respectively.

An example *preventative* approach is as follows. Before removing the connection between k and m, we can check that removing this connection does not disconnect the network before committing the change. Now, to check if removing a connection between agent k and m disconnects the network, some form of network traversal search is needed to determine if agent k can reach agent m without traversing the direct connection between k and m. That is, we need to check if an alternative path exists. Though this approach is effective, it relies on a graph traversal search that is known to be expensive [99]. Moreover, such a search needs to be repeated every time a connection removal is considered.

A *remedial* approach is one that can recognise, after the fact, that the network of agents has been disconnected, and can then reconnect the network. However, it is not possible to create a remedial approach without making using of a central registry. For example, suppose that the connection between agents m and k has been removed. At this point the network is disconnected, but this has not yet been detected by any agent. The next time a service needs to be located it will potentially be unreachable, so when agent m requires a service that only q and n offer, it cannot find it. If we assume that at least one instance of the service exists then, when it cannot be located, we can only assume that the network is disconnected. The only way the network can be reconnected, and the service located, is through a central registry. As we are aiming to use a decentralised approach, reintroducing a central registry is not a suitable option.

In consequence of this discussion, both preventative and remedial approaches weaken the value of the solutions we aim to produce. Moreover, by spending significant time on maintaining or repairing the network, we may potentially hide the destructive effect of an approach that continuously disconnects the network. For these reasons, we choose not to avoid disconnection, but instead to allow tasks to fail if an agent cannot locate a required service.

Importantly, determining when a service location process has actually failed is difficult. If a search process has continued for a length of time, it is impossible to distinguish a search process that has not searched far enough, and a search process that will never locate the service due to disconnection. As we have done previously, to address this in our work, we limit the depth of *network based search*. If this limit is reached, and the service has not been located, then the search fails, as does the task.

7.4 Experimentation and Results

In evaluating our structural adaptation approaches, we are concerned with dynamism. In this context, we cannot use centralised service location, but must instead use *network based search*. Importantly, since a *fully connected structure* cannot be changed, and since *pipelines* and *hierarchies* will immediately be broken by the adaptation, the experiments in this section consist of agents connected by a *random structure*, with *network based search*, and either *random*, *full service connection*, or *frequent service connection* reorganisation. All other aspects of the experiments are as specified in the previous chapters, using the experimental setup described in Chapter 4, and the experimental parameters described in Section 4.4. All experimental results are plotted as averages over time. Where appropriate we have endeavoured to include error bars to show the distribution of the results. However for brevity and ease of understanding we have, in some cases, chosen to merge the results from multiple experiments into a single summary graph and excluded error bars. In such cases, the fully plotted results, along with error bars, can be seen in Appendix B.

To make the effect of structural adaptation clear, in each experiment, no structural adaptation is used until time step 999. Structural adaptation is then used from time step 1000 onwards. Results that are not plotted over time (that is to say, the frequency distribution of *service location time*), only include data collected *after* structural adaptation begins, so that our results reflect the performance of structural adaptation alone.

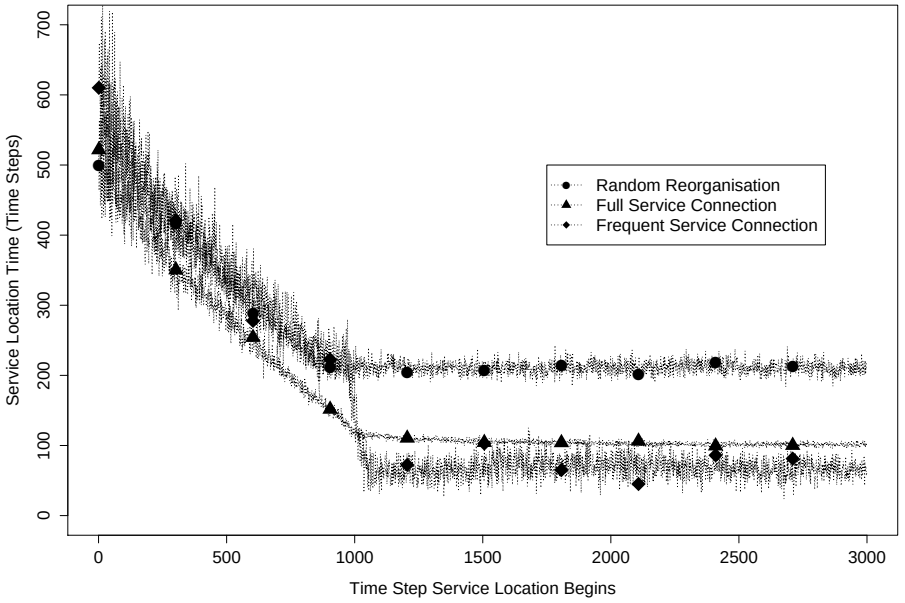


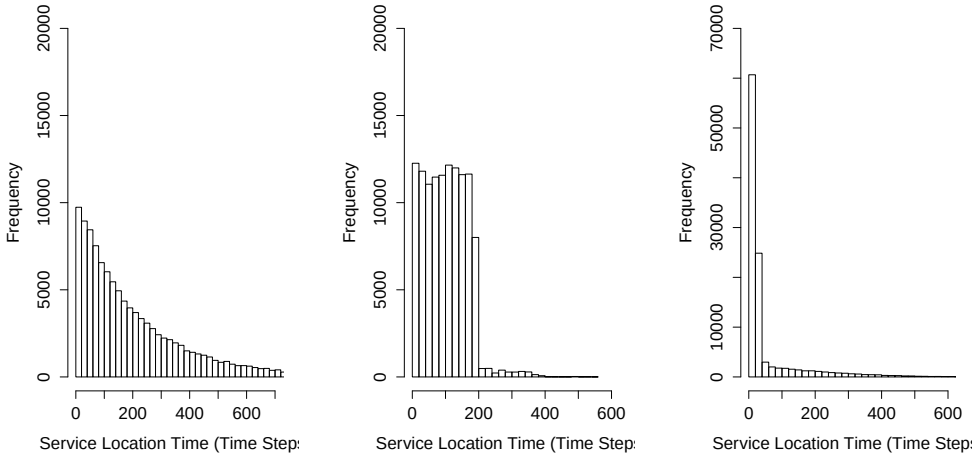
FIGURE 7.2: Service location time for structural adaptation

Additionally, since Chapter 6 identifies a relationship between the number of connections in an organisation's structure, the structure's degree distribution, and *service location time*, we consider the number of connections and *degree distribution* where appropriate, in addition to *service location time* and *waiting for capacity time* as before.

7.4.1 Random Reorganisation

Our first experiment consists of a set of agents employing *random reorganisation* to adapt their connections. When random changes are made to a *random structure*, we expect no impact on *service location time*, compared to when the same structure is static. This is because making a random change to a *random structure* produces another *random structure*.

Figure 7.2 shows that for a service location process that begins at time step 1, *service location time* is 500 time steps. At time step 1000 (when reorganisation begins), *service location time* is lower, at approximately 210 steps (highly significant with p-value 5.19×10^{-20}), showing that, in fact, *service location time* improves. We have previously shown that this is typically due to an increase in the structure's total number of connections, or an increase in its *degree distribution*, but this is not the case here. As



(a) Random reorganisation (b) Full service connection (c) Frequent service connection

FIGURE 7.3: Service location time frequency distribution for structural adaptation

the structure is random, and changes are random, the number of connections remains constant as does *degree distribution*.

This decrease in *service location time* is instead due to the continual changes being made to the structure, which occur throughout the service location process. By changing an agent's connections, entirely new branches of the network are opened up to the service location process, so that more agents are queried in a shorter space of time.

In addition, between time steps 1 and 1000, *service location time* gradually decreases from 500 to around 210 time steps. Though it appears that *service location time* is somehow being affected before structural adaptation begins, this decrease is in fact due to the length of time that it takes to locate services across a static structure. Consider, for example, a service location process that begins at time step 1, and takes 1200 steps to locate a service. This service location process will only benefit from structural adaptation for the last 200 time steps, so will perform poorly. Now, consider instead the same search process beginning at time step 500. Here, it can benefit from structural adaptation much earlier, so *service location time* is decreased. Finally, if the same process begins at step 1000, then it benefits from the effect of structural adaptation immediately. This causes the gradual decrease observed in Figure 7.2.

Figure 7.3(a) shows that the frequency distribution of *service location time* exhibits a tail-off effect. This is because as *network based search* progresses (searches further afield), the time needed to query additional agents increases exponentially.

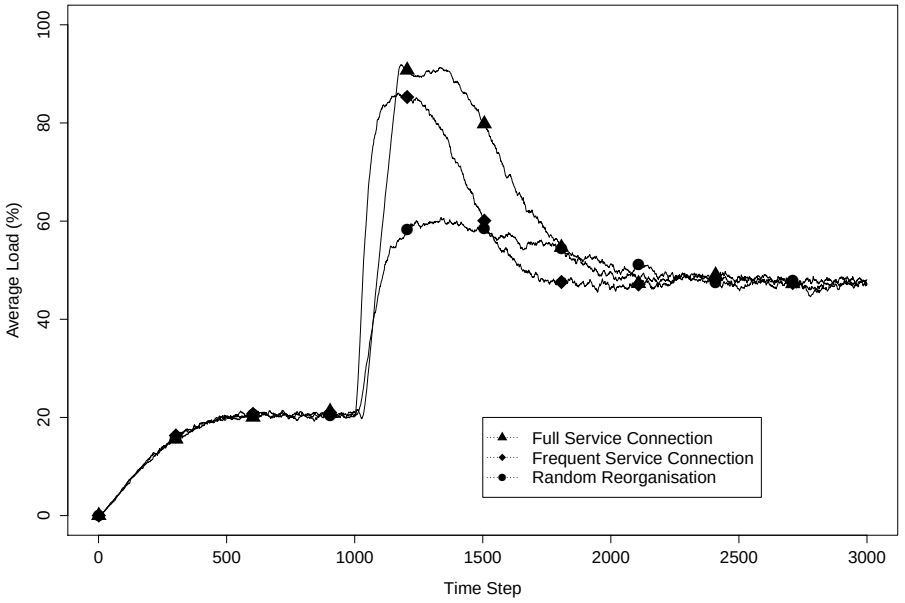


FIGURE 7.4: Average load for structural adaptation

Importantly, Figure 7.4 shows that when structural adaptation begins, agent load almost immediately increases. This is because by this point *service location time* has decreased, so more services are being located, and allocated, causing agents to have more tasks to execute. More specifically, there is an initial rapid increase in load, followed by a slight decrease that eventually reaches a plateau. The initial increase beyond the plateau is caused by the relatively poor performance before structural adaptation begins. Between steps 1 and 1000 it takes a long time to locate services, so there is a backlog of tasks waiting to be executed. However, once structural adaptation begins, *service location time* decreases significantly, so now all tasks (newly arrived tasks, and backlogged tasks) are rapidly allocated to agents for execution, increasing load. Once the backlog has finished, and only new tasks are being allocated, the initial increase in load decreases slightly.

In considering the number of overloaded agents, Figure 7.5 shows that when the backlog of tasks begins to be executed, a maximum of 40% of all agents are overloaded, which decreases to 15-25% after the backlog of tasks is completed.

Finally, Figure 7.6 shows that *waiting for capacity time* increases when structural adaptation begins, but then begins to decrease, and continues to decrease when the experiment ends. This is due to the initial number of tasks being allocated once structural

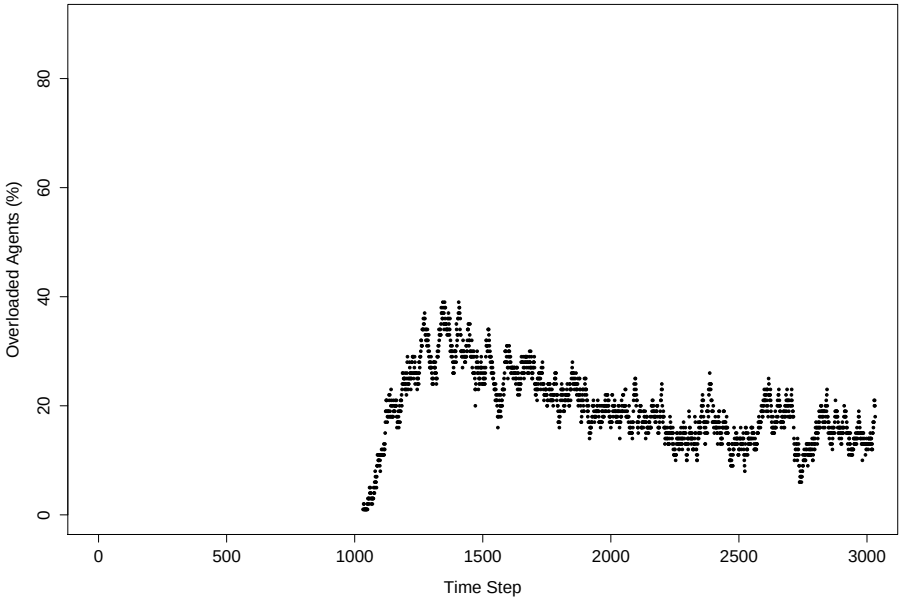


FIGURE 7.5: Random reorganisation: number of overloaded agents

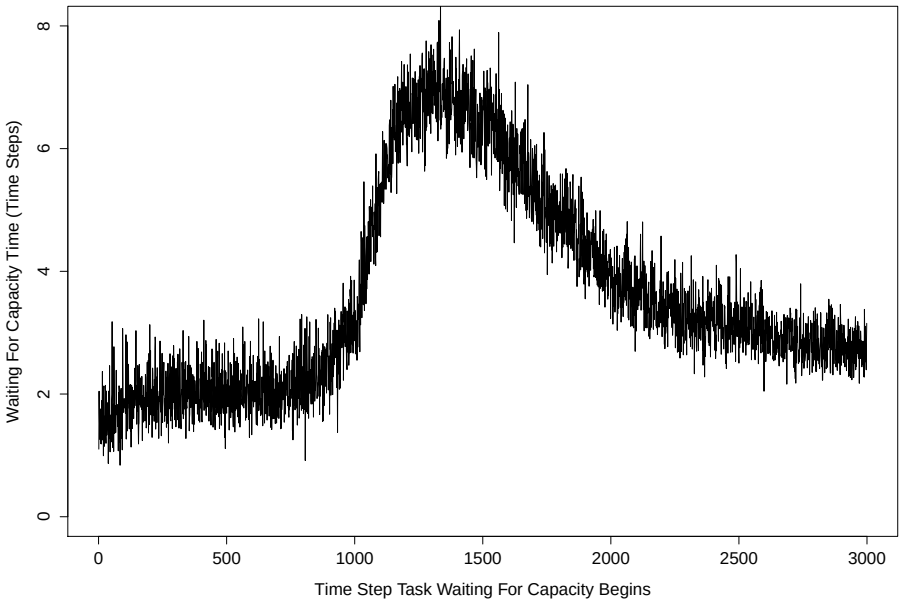


FIGURE 7.6: Random reorganisation: waiting for capacity time

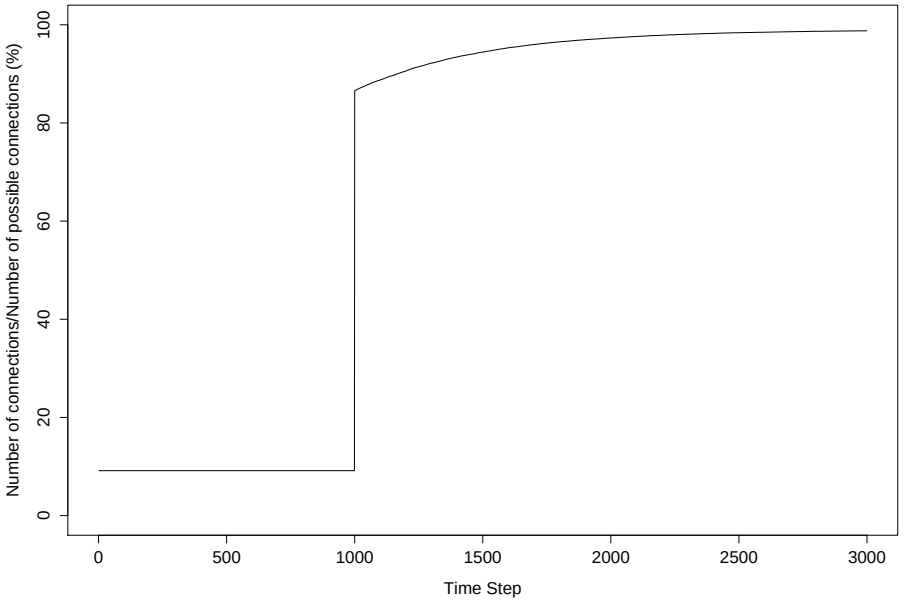


FIGURE 7.7: Average number of connections: full service connection

adaptation begins, and to both new tasks and the backlog of tasks being allocated. As with agent load, once the system has caught up with the backlog of tasks, *waiting for capacity time* decreases.

To summarise, continual, arbitrary changes to an organisational structure can decrease *service location time*. This is not because the total number of connections in the structure have been increased, nor is it because any single agent has more connections at any point in time. Instead, by continually changing connections, an agent has an increased number of connections when considered over a time period. Because service location takes time, during that time the number of agents that can be directly contacted is larger than the number of connections at any fixed point in time.

7.4.2 Full Service Connection Reorganisation

In experimenting with all agents using *full service connection* reorganisation, Figure 7.2 shows that *service location time* is almost as low as that of the *fully connected structure* in Chapter 6. This is because when structural adaptation begins, a large number of connections are created very quickly. As can be seen in Figure 7.7, within a few time steps approximately 90% of all possible connections have been created and, by the end

of the experiment, a *fully connected structure* has emerged. Thus, with regard to *service location time*, *full service connection* reorganisation performs as well as a *fully connected structure*, because it produces a *fully connected structure*.

Soon after *structural adaptation* begins, the average load of the system rapidly increases, as shown in Figure 7.4. This increase exceeds that for *random reorganisation*, and it also decreases faster to the same plateau. The load eventually achieved is the same as with a *fully connected structure*. The number of overloaded agents initially rises, but at a much higher rate than with random reorganisation (increasing to 80% of agents being overloaded, rather than 35–40% previously) (Figure 7.8). After this initial increase, the number of overloaded agents falls to that of the *fully connected structure*. The same initial increase can be seen in Figure 7.9, which shows *waiting for capacity time* initially increasing, but then dropping to the same level as *fully connected structure* as the backlog of tasks is completed.

To summarise, *frequent service connection* can improve *service location time*, though this achieved by effectively creating a fully connected structure. However, this is due to the rarity of instances of particular services. We can imagine that in a scenario where there are very few services and many agents offering instances of the same services, then similar results can be achieved, with a reduced number of connections.

7.4.3 Frequent Service Connection Reorganisation

Finally, when agents adapt their connections using *frequent service connection* reorganisation, as shown in Figure 7.2, average *service location time* decreases to between 60 and 75 time steps. When *full service connection* is used, *service location time* is improved by increasing the total number of connections in the structure, but by examining the results in Figure 7.10 we can see that this is not the case here. To improve *service location time*, *frequent service connection* does increase the number of connections, yet by the end of the experiment only 18% of all possible connections are created, compared to 100% when *full service connection* is used. Essentially, *frequent service connection* improves performance compared to *full service connection*, while using fewer connections. This is a highly significant improvement with a p-value of 3.90×10^{-27}. This increase in performance can be partially attributed to the marginal increase in the number of connections, and the marginal decrease in the distance between agents and the services they require. However, the major contributing factor is that regularly used services are

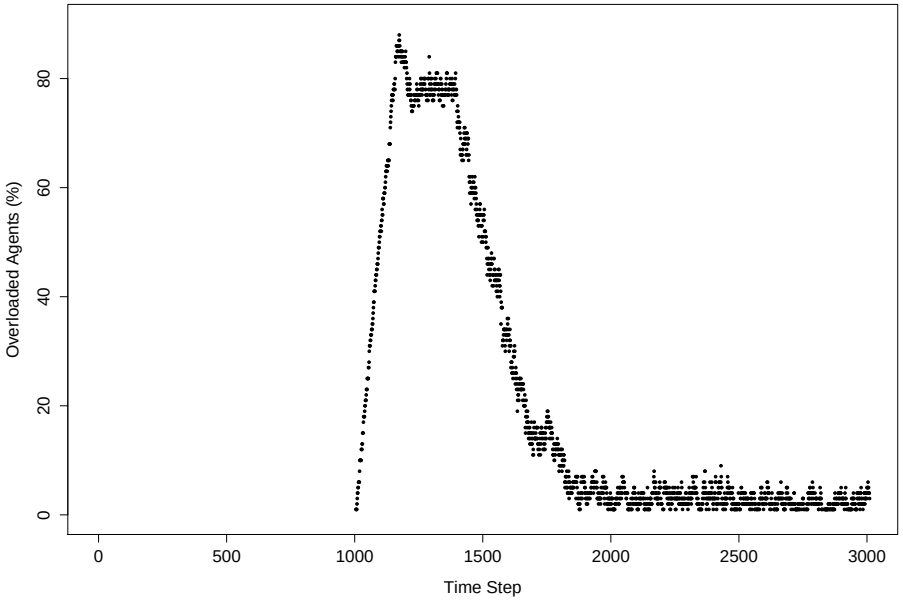


FIGURE 7.8: Number of overloaded agents: full service connection

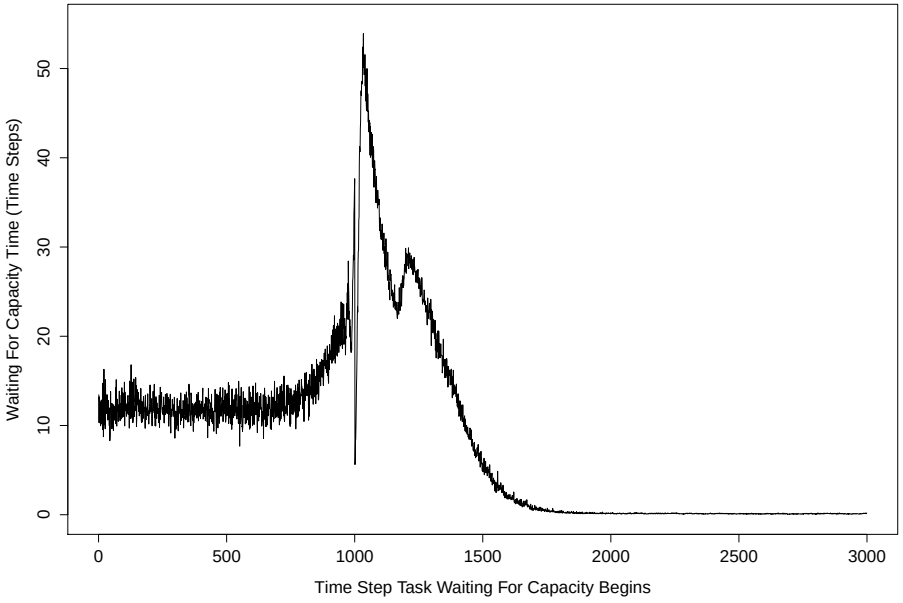


FIGURE 7.9: Average waiting for capacity time: full service connection

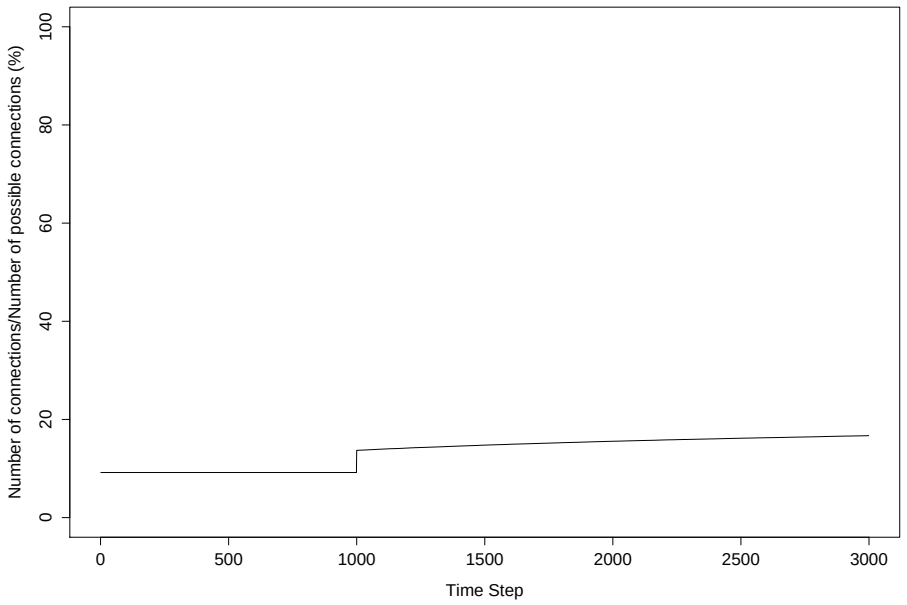


FIGURE 7.10: Average number of connections: frequent service connection

much closer, and rarely used services are further away. Less frequently used services are harder to locate, but as they are rarely needed, the overall result is improved *service location time*.

To test whether these results are dependent on the number of agents in the system, we repeated this experiment with 25, 50, and 75 agents. Figure 7.11 shows that as the number of agents decreases, average service location time also decreases, and this is expected — the fewer agents there are, the easier it is to locate services. At time step 1000, when reorganisation begins, we can see that regardless of the number of agents, the same pattern of improved service location time is seen. We also repeated the experiment for 50, 100, 150, and 200 services, and Figure 7.12 shows that regardless of the number of services, the same pattern of improved service location time can be seen when reorganisation begins.

Though, on average, *service location time* decreases from 100 time steps to approximately 70, Figure 7.3(c) shows that after 20 time steps, approximately 90% of all service location processes have successfully located an appropriate agent. This is a big improvement for most cases, but for services that are rarely used, service location can take much longer, in some cases up to 3500 time steps. In addition, the use of this approach causes

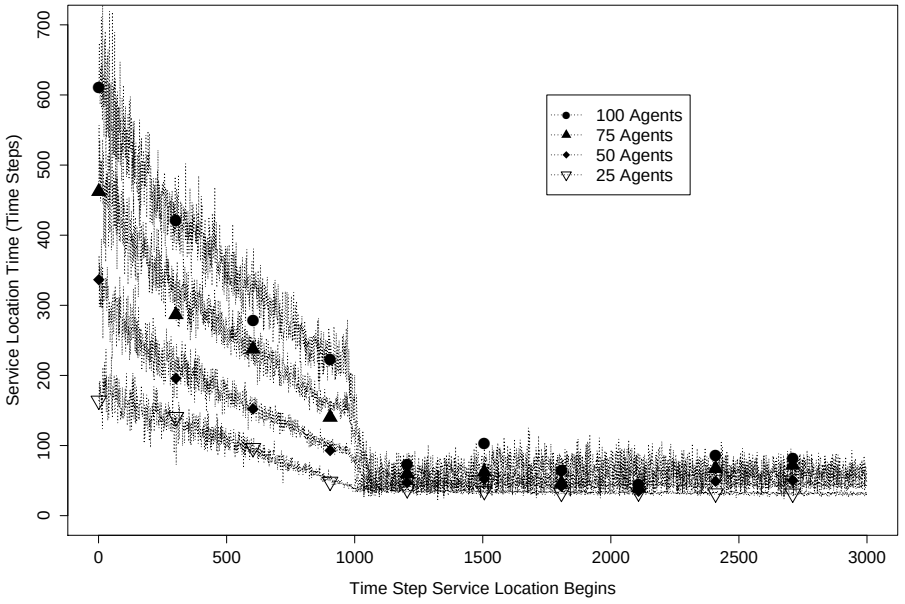


FIGURE 7.11: Average Service Location Time: Frequent Service Connection, varied number of agents

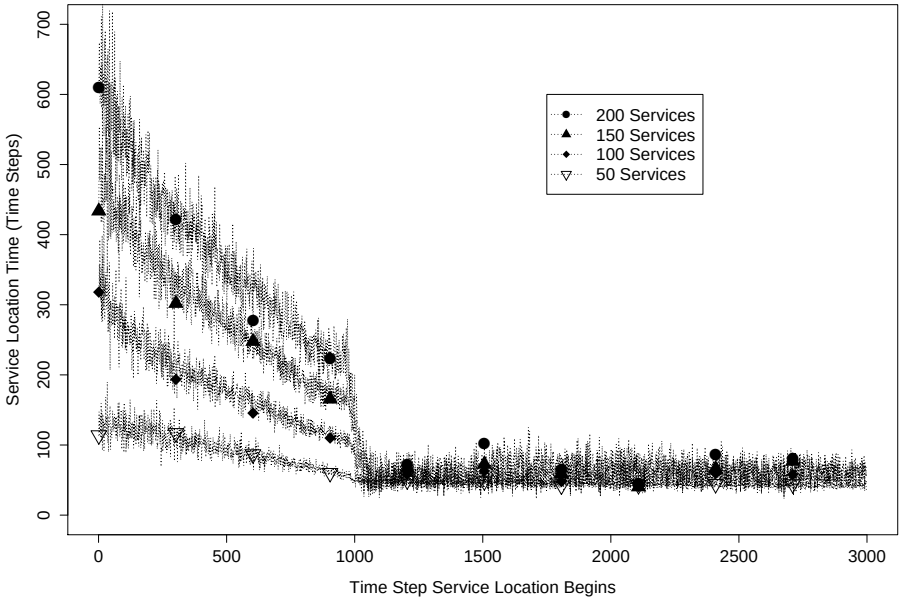


FIGURE 7.12: Average Service Location Time: Frequent Service Connection, varied number of services

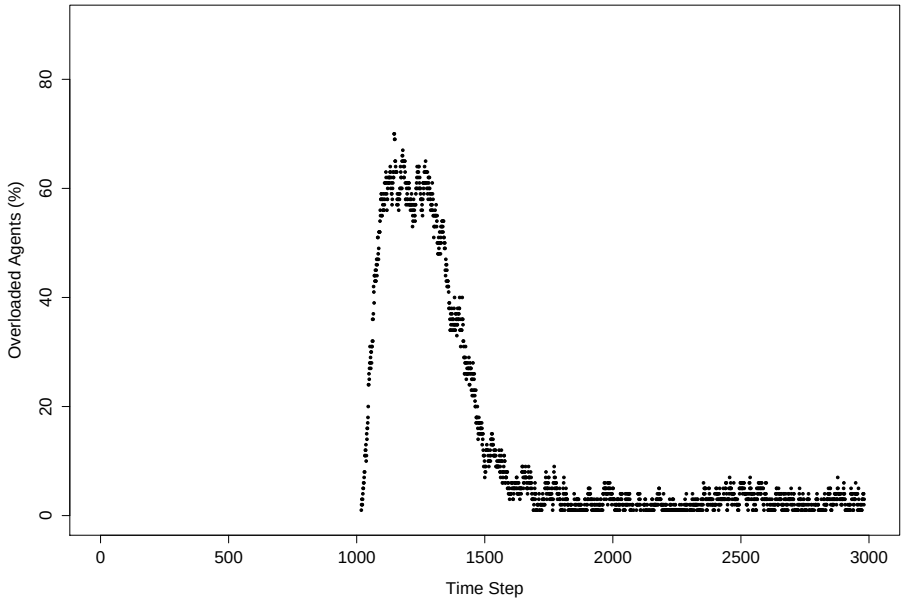


FIGURE 7.13: Number of overloaded agents: frequent service connection

approximately 0–0.1% of tasks to fail. This is an improvement on the number of failed tasks caused by a random structure with no reorganisation, but is a weakness when compared to a central registry, which manages to locate services for all tasks.

Like the other reorganisation approaches, *frequent service connection* exhibits an initial increase in agent load, as is evident from Figure 7.4, but this increase occurs sooner because services are located faster. In addition, it also catches up with the backlog of tasks quicker. Figure 7.13 shows that the number of overloaded agents is initially zero, but when reorganisation begins the number of overloaded agents jumps because of the increase in the number of tasks being allocated. However after approximately 500 time steps the backlog of tasks is complete and the number of overloaded agents is similar to that of a centralised or fully connected system.

The same trend is shown in Figure 7.14, where the average *waiting for capacity time* initially increases, but eventually drops to the same level as the traditional systems. It is interesting to note the two peaks in *waiting for capacity time*, which are caused because services are located so quickly. First, the services for 90% of all backlogged tasks are located within a 20 time step window, and are all allocated within that window, causing the first peak. Now, recall from Chapter 3 that all tasks have an *initial requirement*,

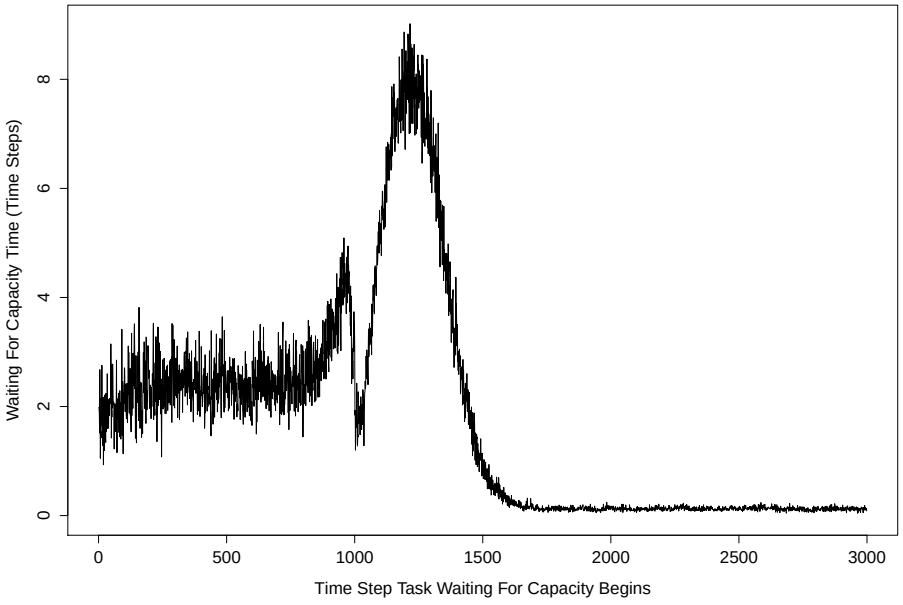


FIGURE 7.14: Average waiting for capacity time: Frequent service connection

potentially followed by multiple other requirements. In this first peak, these initial requirements are executed, and *waiting for capacity time* decreases once they start to be completed. However, subsequently, the requirements of any subtasks must then be executed and allocated; since service location is quick, these cause the second peak in *waiting for capacity time*.

In summary, for an environment in which there exists some correlation between service requirements, *frequent service connection* can achieve a dramatic performance increase in average *service location time*, by decreasing the distance to regularly used services, and increasing the distance to rarely used services. However, a small number of services fail to be located, so if failure to locate services is unacceptable, or if there is no correlation between service requirements, then *full service connection* is a better option, increasing *service location time*, but never failing, and there is no tail-off effect for the service location time frequency distribution.

7.5 Discussion

In Section 2.4.6 we reviewed existing evaluations, and found them to be very broad, or extremely specific to a particular case. Through our evaluation, we have provided an analysis that is broad enough that we understand the full impact of structural adaptation, and is also specific enough to elicit more from the results than a simple increase or decrease in global performance. In addition, by offering a comparison between *full service connection*, *frequency service connection*, and an approach that makes random changes to the structure, we can also distinguish the benefit of making random changes from the benefit produced through agents actively determining which connections to change. In general, we have found that all forms of change can improve task completion time and, in particular, service location time. However, with regard to *frequent service connection*, service location can on rare occasions fail, causing tasks also to fail. This is typically not an acceptable outcome for any system, but there are some applications where it may be acceptable for some tasks to fail, to achieve significant performance benefits.

Such behaviour may be acceptable for any application where the result converges to the actual value as the sample rate is increased, so that the more samples that are produced, the more accurate the result. As each sample is usually calculated independently, any Monte Carlo approach that uses random sampling is a perfect candidate to be distributed across a number of agents, so that each agent calculates a set of samples before they are all aggregated. Now if calculating a sample, or even a set of samples, is a single task in our experiments, then a failed task causes the sample rate to be decreased, but does not cause the overall Monte Carlo simulation to fail. If the number of samples is decreased significantly, then this will affect the accuracy of the result, but as the *frequent service location* approach causes only 0–0.1% of tasks to fail, this produces a 99.9% sample rate. This decrease is unlikely to have a significant effect on any results, and if it does then some additional samples can be calculated at little extra cost. The Monte Carlo method is used in many domains, such as in bioinformatics for phylogenetic inference [4], financial engineering for derivative pricing [45], statistical physics [12], and social sciences [83].

7.6 Conclusion

To summarise, by simply locating services in a decentralised manner, we can cause service location to take significantly longer than centralised systems, making it difficult to even assess the allocation and execution of tasks past the point of locating services, as shown in Chapter 6. However, we have shown in this chapter that by applying any type of *structural adaptation*, be it *random*, *full service connection*, or *frequent service connection*, *service location time* can be significantly improved, and the majority of services can be located.

Random reorganisation can improve *service location time*, meaning that change, for its own sake, can improve system performance. However, *service location time* can be further improved by using *frequent service connection*, which improves *service location time* by increasing the total number of connections in the structure. Finally, *frequent service connection* can improve average *service location time* further still by crucially decreasing the distance between agents and their most frequently required services.

Though *frequent service connection* reorganisation performs well, a small number of tasks always fail. In general, this is considered to be a bad property of any system, particularly any system where robustness is a key requirement, but in some contexts it can be an acceptable property given the performance benefits it achieves.

Chapter 8

Structural Adaptation Constrained By Topology

8.1 Introduction

While we have developed techniques that adapt an organisational structure to improve task throughput, these pay little attention to the underlying need for any particular structure to suit the application at hand. As we noted earlier, particular structures can be seen in many applications, in the organisations, tasks, and the protocols that are used. In this chapter, we extend our previous work to include consideration of topological constraints by introducing techniques to maintain these constraints while adapting an organisational structure.

In the context of our work, a set of structural constraints is known as a *topology*, which defines the allowed connections in an organisational structure. The constraints ensure that the structure follows some pattern, such as a hierarchy following a tree structure, or a pipeline following a line structure. In static organisations, the initial formation of the organisation must ensure that any connections created follow a given *topology* that is usually specified at design time. However, when the organisational structure is adapted at runtime, as in our approaches, steps need to be taken to ensure that the *topology* is not broken when changes are enacted.

Topology preservation is not trivial, as it requires substantial knowledge, not just about a single agent's connections, but also about the organisational structure around it; this

is information that an agent may not have, or cannot easily obtain. The complexity of preserving topology is amplified when multiple agents simultaneously make changes to the structure and, while each individual agent's changes may not break the topology, when combined they may violate the topological constraints. While we assume that all agents perform task allocation and execution autonomously, in order to address the complex problem of topology preservation, we introduce global knowledge and control of the organisational structure to ensure that any adaptation performed preserves the organisation's *topology*.

This chapter is structured as follows. In Section 8.2 we discuss how topology can be preserved while reorganising, by filtering changes before they are enacted. We then discuss more specifically how we can ensure that one change to a pipeline, hierarchy, or other topologies, can be enacted while preserving topology, in Sections 8.3, 8.4, and 8.5, respectively. In Section 8.6 we then show how the same can be achieved for multiple changes. Finally, we evaluate the performance of reorganisation while preserving topology in Section 8.7, and conclude the chapter in Section 8.8.

8.2 Filtering Changes

Consider the *frequent service connection* approach of Chapter 7, which can analyse the frequency at which services are required, regardless of whether an agent is in a hierarchy, pipeline, or a random structure. The result of this approach, and others like it, is a structure that is continuously adapted. In this section, we reconsider this approach and break it down into component parts, so that changes are initially suggested and subsequently filtered in order to consider the impact on topological constraints. Thus, when reorganising, certain structural changes are not possible. To ensure that these topological constraints are preserved, we need to ensure that no enacted changes, or sets of changes, will violate these constraints.

In adapting a structure, we can ensure that topology is preserved in one of two ways. First, we can use a preventative approach, by ensuring that when a set of adaptations to a structure is proposed, nothing is changed unless the resulting structure maintains the existing topology. However, restricting individual adaptations as they are generated can be excessive in ruling out potential solutions. For example, if the smallest adaptation that preserves a particular topology always consists of at least two changes, such as

removing a connection and adding a connection, but individually these are not permitted, then a single adaptation can never be considered, so reorganisation is impossible.

One change may break a topology, but subsequent changes may break and then re-establish the topology, ensuring that it is maintained while giving greater freedom when reorganising. Therefore, the second and preferred option is to use our reorganisation approaches as they have previously been described, ignoring any topological constraints, and instead of enacting any changes, producing an initial set of changes — an *initial change set*, ICS — that could break the topology. This *initial change set* can then be adapted to produce a *final change set*, FCS, which is a set of changes, based on the *initial change set*, that preserves the current topology. In this way, reorganisation is a more modular process with a separation between the reorganisation and preservation of topology.

We can see this in Algorithm 5, where *mas* is a multiagent system being reorganised, *suggestStructuralChanges* is a function that takes a multiagent system and creates an *initial change set* according to some analysis it performs, and *preserveTopology* is a function that takes an *initial change set* and the *mas* to which it applies, and produces a *final change set* that adheres to topological constraints. The overall reorganisation process is known as *topology preserving reorganisation*, which is illustrated in Figure 8.1. Instances of the function *suggestStructuralChanges* have already been specified in Chapter 7, like *full service connection* and *frequent service connection*. The function *preserveTopology* is discussed in more detail in the rest of this chapter.

Algorithm 5 *topologyPreservingReorganisation*(*mas*)

1. ICS $\leftarrow$ *suggestStructuralChanges*(*mas*)
2. FCS $\leftarrow$ *preserveTopology*(ICS, *mas*)
3. *enact*(FCS, *mas*)

8.2.1 Change Sets and Legality

The smallest adaptation that can be made to any structure is either to add or remove a connection between two agents, which we refer to as a *change*. If multiple *changes* are to be performed together then they can be grouped into a *change set*. Removing a particular connection, or creating another, may break certain topological constraints, but additional *changes* may then re-establish the topology. If a *change set* preserves

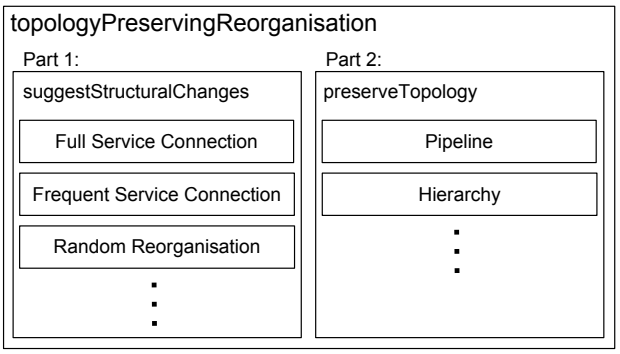


FIGURE 8.1: Summary of *topologyPreservingReorganisation*.

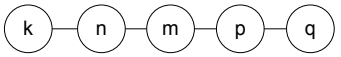


FIGURE 8.2: An example pipeline.

topological constraints, even if this is by breaking and re-establishing the topology, then this change set is *legal*.

When there is no topology to preserve, any connection can be removed or added, and so every *change set* is *legal*, even those that contain one *change*; otherwise, however, *legal change sets* are more complex. To illustrate, the connection between agent k and agent n in Figure 8.2 cannot be removed without breaking the pipeline topology. However, if a connection is removed between k and n, and then a connection is created between k and q, the topology is preserved.

Generalising this, we can define a *change set* as a set of the form $\text{CS} = \{cc_1, cc_2, \dots\}$, where each element is single *change*, represented as $cc = (a_1, a_2, act)$, where, for a change cc, act takes one of the values in the set {*'remove'*, *'create'*}, which respectively denotes the action of *removing* or *creating* a connection between agents a_1 and a_2. Thus, in Figure 8.2, the change sets $\{(k, n,$ *'remove'*$)\}$ or $\{(k, q,$ *'create'*$)\}$ are individually not *legal* with respect to the topology, but if both changes are performed as the change set $\text{CS} = \{(k, n,$ *'remove'*$), (k, p,$ *'create'*$)\}$, then this *change set* is *legal*.

8.2.2 Transformations and Transformation Templates

If our intention is to remove the connection between agents k and n while maintaining the topology then, given that the change set $\text{CS} = \{(k, n,$ *'remove'*$), (k, p,$ *'create'*$)\}$ is *legal*

with respect to the pipeline in Figure 8.2, this is one of many possible ways in which this can be achieved, and we refer to this as a *transformation*. A *transformation* is a change set that contains at least one change, will maintain a structure's original topology, and contains no more changes than required to ensure that topology is maintained. That is, a *transformation* is a non-empty change set that is legal, and also has the fewest changes possible — it is *minimal*. When no topology needs to be preserved, a *transformation* always consists of a single change. However, when a topology must be maintained, a *transformation* typically consists of at least two changes.

To specify a *transformation* for a particular topology without specifying which agents are involved in the *transformation*, we can define a *transformation template*, which consists of: some changes, where the agents in each change are variables; and some constraints between these agents, such as their positions in relation to each other, or each agent's number of connections. A *template* can then be used to generate a *change set* that is guaranteed to be *legal*, as follows. Suppose we have a single change, such as $cc = \{(k, q, \text{'create'})\}$ from the pipeline example discussed above, which is not *legal* on its own, but can be legal as part of a larger change set. By instantiating the agent variables in a *transformation template* with the specific agents from change cc, we can instantiate all other variables for the remaining changes, and so generate a *change set* that is legal.

8.2.3 Initial and Final Change Sets

By employing *transformation templates*, we can generate a *change set* that contains a given change, and is *legal* — it will preserve topology. However, each of our reorganisation approaches (*full service connection*, *frequent service connection*) does not *suggest* a single change, but instead *suggests* multiple changes in the form of an *initial change set*, ICS. Our aim, then, is to use *transformation templates* to produce a *final change set*, which contains changes from the *initial change set*, and is *legal*.

Ideally, we want the *final change set* to include all changes from the *initial change set*, but in some cases the topology will restrict this. Consider a scenario in which an *initial change set* recommends three changes to the pipeline in Figure 8.2: create connections between k and m, m and q, and q and k. Regardless of the structure before enacting these changes, a loop is always formed between agents k, m, and q, breaking the topology, so it is impossible to include all of the changes. If these are our initial changes, then in

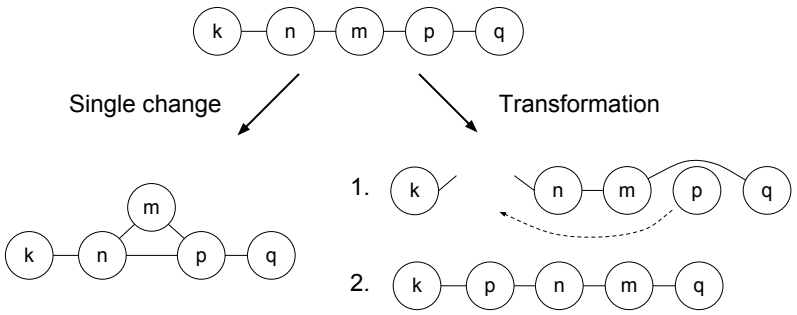


FIGURE 8.3: An example of enacting a change in a pipeline.

seeking to preserve topology we should aim to include as many changes as possible from this *initial change set*.

Broadly, we can generate a *legal change set* for each element in the *initial change set* by instantiating a *transformation template* and then, by combining each of these change sets, we can produce a *final change set*. However, combining these sets is not trivial, because there may be conflicts between them, which must be resolved.

In what follows, we consider different topologies and the *transformations* that can be applied to them, while maintaining topological constraints. We then generalise each of these *transformations* to a number of *transformation templates* that can later be used to generate *transformations* for any structure. We consider the preservation of *pipelines* and *hierarchies*, in Sections 8.3 and 8.4, respectively, with additional discussions about *rings*, *stars*, and a *transformation* that applies to all structures regardless of topology in Section 8.5. This is followed by a description of how *transformation templates* can be used to generate a *final change set* in Section 8.6.

8.3 Pipelines

In a multiagent system that is linked by a pipeline topology, all agents are aligned sequentially. Each agent has two connections, except for the agents at the ends of the line, which have exactly one. Formally, this can be represented as follows, where

$degree(a)$ represents agent a's number of connections.

$$pipeline(mas) \implies \begin{array}{l} mas = (\text{A}, \text{C}) \wedge \\ \exists a_2, a_3 \in \text{A}\ \forall a_1 \in [\text{A} \setminus \{a_2, a_3\}] \\ degree(a_1) = 2 \wedge \\ degree(a_2) = 1 \wedge \\ degree(a_3) = 1 \wedge \\ a_2 \neq a_3 \end{array}$$

This can be understood to mean that, apart from agents a_2 and a_3, which are each connected to one agent, all other agents a_1 in the multiagent system are connected to two agents.

As the constraints above clearly show a fixed number of connections, a single *change* — a single connection added or removed — will always break the topology. To illustrate, suppose we have five agents that are in a pipeline structure as shown at the top of Figure 8.3, and a single *change* specifying that a connection should be created between agents p and n, $(p, n, \text{`}create\text{'})$. As can be seen on the left hand side of Figure 8.3, if this *change* is enacted, then the pipeline topology is not maintained, so the *change* is not *legal*. In this context, in a pipeline, every time one connection is removed another connection must be created, and likewise every time a connection is created another must be removed.

8.3.1 Pipeline Transformations

A single change can never be enacted on a pipeline. Instead, to ensure that a pipeline topology is maintained we must enact a *transformation* which, as stated previously, is a *change set* that is *legal* and *minimal*. If we consider the agents in a pipeline to be a list or queue, then a single *transformation* is one that reorders the agents in the list such as moving an agent forward or backward. More specifically, this can be divided into two processes: remove agent a_1 from the pipeline, and reinsert it next to another agent a_2. This involves removing the current connections of agent a_1, and adding new connections so that it is next to agent a_2. This *transformation* is shown on the right hand side of Figure 8.3, where agent p is removed from the pipeline, and the gap it leaves is closed by creating a connection between agents m and q. Agent p is then reinserted into the

pipeline between agent k and n. This transformation is formally represented as follows.

$$\begin{aligned} \text{CS} = \quad & \{(p, m, \text{`remove'}), (p, q, \text{`remove'}), (k, n, \text{`remove'}) \\ & (m, q, \text{`create'}), (p, k, \text{`create'}), (p, n, \text{`create'})\} \end{aligned}$$

Now, through enacting this transformation, we can change the structure in Figure 8.3, and the pipeline is preserved. Alternatively, we can consider this *transformation* as a *legal change set* that contains our original change, $(p, n, create)$, and so we can enact the change on the left hand side of Figure 8.3, while ensuring that topology is preserved.

8.3.2 Generating Transformation Templates

Only one possible *transformation* is described above, but there are actually many possible *transformations* that can be employed, depending from where an agent a_1 is removed, and where it is reinserted. For example, agent p could have been reinserted between agents n and m, instead of k and n, or n could have been removed instead of agent p, and reinserted between agents p and q, all of which result in p being connected to agent n. In addition, the removal and reinsertion of an agent becomes more complicated when considering that either one or both of the agents involved may be at the end of the pipeline. To capture all of these possible *transformations*, we consider the problem again more generally, with the aim of producing some *transformation templates*. Formally, a pipeline *transformation* is the removal and reinsertion of an agent a_1 next to an agent a_2, taking into consideration the original position of agents a_1 and a_2.

Remove Agent Removing a_1 from a pipeline depends on a_1's position in the structure. If a_1 has one connection, then it is at the end of the pipeline, and so its only connection is removed. If a_1 has two connections, then it is in the middle of the pipeline, so both of its connections are removed. In addition, a connection is created between a_1's previous neighbours, ensuring that the pipeline structure is maintained. In both instances the result is a pipeline, excluding the single disconnected agent a_1.

Reinsert Agent Reinserting a_1 next to a_2 similarly depends on a_2's position. If a_2 has one connection, then it is at the end of the pipeline, so a connection is created between a_1 and a_2, and a_1 is now at the end of the pipeline. If a_2 has two connections, then it is in the middle of the pipeline, so we must decide on which side of a_2 to reinsert a_1. To do so, we randomly select one of a_2's neighbours, a_3,

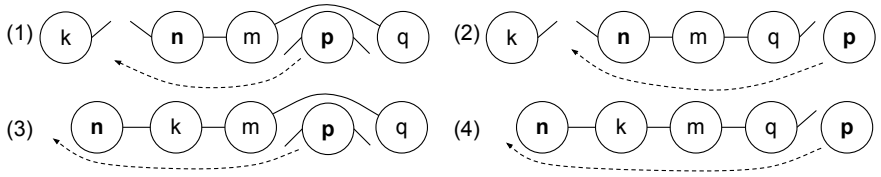


FIGURE 8.4: An example of enacting a change on multiple pipelines.

remove the connection between a_2 and a_3, and create connections between a_1 and a_2, and between a_1 and a_3. In both instances, the result is a pipeline where a_1 is directly connected to a_2.

From the description above, it is clear that the removal and reinsertion of a_1 is dependent on one of four possible states, defined by agent a_1's number of connections, and the same for agent a_2. These four states can be summarised, as shown below, where $degree(a_1)$ represents agent a_1's number of connections.

1. $degree(a_1) = 2 \wedge degree(a_2) = 2$
2. $degree(a_1) = 2 \wedge degree(a_2) = 1$
3. $degree(a_1) = 1 \wedge degree(a_2) = 2$
4. $degree(a_1) = 1 \wedge degree(a_2) = 1$

Examples of the removal of agent p and its reinsertion next to agent n in these four states can be seen in Figure 8.4, where sub-figures (1) to (4) correspond to the four possible states. For each of the *transformations* shown in Figure 8.4, the *change sets* (CS) are as follows.

1. CS $= \{(p, m, \text{'remove'}), (p, q, \text{'remove'}), (k, n, \text{'remove'}),$
   $(m, q, \text{'create'}), (p, k, \text{'create'}), (p, n, \text{'create'})\}$
2. CS $= \{(p, q, \text{'remove'}), (k, n, \text{'remove'}), (p, k, \text{'create'}), (p, n, \text{'create'})\}$
3. CS $= \{(p, q, \text{'remove'}), (p, m, \text{'remove'}), (q, m, \text{'create'}), (p, n, \text{'create'})\}$
4. CS $= \{(p, q, \text{'remove'}), (p, n, \text{'create'})\}$

So far, all of these change sets are *legal* — they preserve topology. However, under specific conditions, they are not always *minimal*. Suppose we have the initial structure shown at the top of Figure 8.3 and, as before, we remove agent p from the pipeline, but instead of reinserting agent p between agents k and n, we instead reinsert p between

agents m and n. Using what we have shown so far, we know that $degree(p) = 2$ and $degree(n) = 2$. Therefore, a possible *legal change set* is as follows.

$$\begin{aligned} \text{CS} \;=\; \{&\mathbf{(p,\ m,\ `remove')}, (p, q, \text{`remove'}), (m, n, \text{`remove'}), \\ &(m, q, \text{`create'}), \mathbf{(p,\ m,\ `create')}, (p, n, \text{`create'})\} \end{aligned}$$

However, this change set has two redundant changes (shown in bold), so it is not *minimal*. This can only occur when agents are exactly a distance of two away from each other, or $distance(p, n) = 2$. In this instance the redundant changes can be removed without affecting the result, producing the following change set.

$$\text{CS} \;=\; \{(p, q, \text{`remove'}), (m, n, \text{`remove'}), (m, q, \text{`create'}), (p, n, \text{`create'})\}$$

For each of the four possible states given previously, the two agents a_1 and a_2 can also be either a distance of two apart, $distance(a_1, a_2) = 2$, or not a distance of two apart, $distance(a_1, a_2) \neq 2$. Therefore, we now have eight possible states. Provided that we have two agents a_1 and a_2 that are currently not connected, and we wish to connect them, we have all the tools we need to infer *transformation templates* that can be instantiated to generate *legal* change sets.

8.3.3 Transformation Templates

Based on the eight states described above, Table 8.1 shows a set of *transformation templates* for pipeline structures (the table assumes that $a_1, a_2, a_3, a_4, a_5 \in \text{A}$, and $a_1 \neq a_2 \neq a_3 \neq a_4 \neq a_5$). Given a single change, such as $(a_1, a_2, \text{`create'})$, an appropriate *transformation template* can be chosen according to the constraints on a_1 and a_2, and this *template* can be instantiated to generate a *transformation* that is guaranteed to be both *legal* and *minimal*, as follows.

Consider again the pipeline at the top of Figure 8.3, and a change $cc = (n, p, \text{`create'})$, where we can instantiate two agents in the following manner: $a_1 = n$, and $a_2 = p$. Then, since $degree(n) = 2$, $degree(p) = 2$, and $distance(n, p) = 2$, we can identify the following *transformation template* in Table 8.1 as being appropriate:

$$\begin{gathered} \{(a_1, a_3, \text{`remove'}), (a_2, a_4, \text{`remove'}), (a_3, a_4, \text{`create'}), (a_1, a_2, \text{`create'})\} \\ \textit{where} \\ d(a_1, a_3) = 1 \wedge d(a_2, a_3) = 1 \end{gathered}$$

Conditions			**Transformation templates**
$degree(a_1)$	$degree(a_2)$	$distance(a_1, a_2)$	
2	2	$\neq 2$	$\{(a_1, a_3, \text{'remove'}), (a_1, a_4, \text{'remove'}),$ $(a_5, a_2, \text{'remove'}), (a_3, a_4, \text{'create'}),$ $(a_1, a_5, \text{'create'}), (a_1, a_2, \text{'create'})\}$
2	2	$= 2$	$\{(a_1, a_3, \text{'remove'}), (a_2, a_4, \text{'remove'}),$ $(a_3, a_4, \text{'create'}), (a_1, a_2, \text{'create'})\}$ where $[distance(a_1, a_3) = 1 \wedge$ $distance(a_2, a_3) = 1] \vee$ $[distance(a_1, a_4) = 1 \wedge$ $distance(a_2, a_4) = 1]$
2	1	$\neq 2$	$\{(a_1, a_3, \text{'remove'}), (a_1, a_4, \text{'remove'}),$ $(a_3, a_4, \text{'create'}), (a_1, a_2, \text{'create'})\}$ or $\{(a_2, a_3, \text{'remove'}), (a_1, a_4, \text{'remove'}),$ $(a_2, a_4, \text{'create'}), (a_1, a_2, \text{'create'})\}$
2	1	$= 2$	$\{(a_1, a_3, \text{'remove'}), (a_1, a_2, \text{'create'})\}$
1	2	$\neq 2$	$\{(a_2, a_3, \text{'remove'}), (a_2, a_4, \text{'remove'}),$ $(a_3, a_4, \text{'create'}), (a_1, a_2, \text{'create'})\}$ or $\{(a_1, a_3, \text{'remove'}), (a_2, a_4, \text{'remove'}),$ $(a_1, a_4, \text{'create'}), (a_1, a_2, \text{'create'})\}$
1	2	$= 2$	$\{(a_2, a_3, \text{'remove'}), (a_1, a_2, \text{'create'})\}$
1	1	$\neq 2$	$\{(a_1, a_3, \text{'remove'}), (a_1, a_2, \text{'create'})\}$ or $\{(a_2, a_3, \text{'remove'}), (a_1, a_2, \text{'create'})\}$
1	1	$= 2$	$\{(a_1, a_3, \text{'remove'}), (a_1, a_2, \text{'create'})\}$ or $\{(a_2, a_3, \text{'remove'}), (a_1, a_2, \text{'create'})\}$

TABLE 8.1: All transformation templates containing a change to connect agents a_1 and a_2 in a pipeline

By instantiating the values for a_1 and a_2 we get the preliminary change set:

$$\{(n, a_3, \text{'remove'}), (p, a_4, \text{'remove'}), (a_3, a_4, \text{'create'}), (n, p, \text{'create'})\}$$

Finally, using the extra conditions that $distance(n, a_3) = 1$ and $distance(p, a_3) = 1$, we can determine that $a_3 = m$, and $a_4 = q$, producing the following change set, which we can guarantee is both *legal*, and *minimal*:

$$\{(n, m, \text{'remove'}), (p, q, \text{'remove'}), (m, q, \text{'create'}), (n, p, \text{'create'})\}$$

A similar process can be used with all *transformation templates* for each topology. In this way, a *transformation* can be created for a single change.

Rings are sufficiently similar to pipelines that they can be considered the same with regard to topological preservation. Though we could analyse each topology separately, and find a set of *transformation templates* to maintain a ring topology, it is easy to see that a ring is simply a pipeline that loops back on itself such that every agent's degree of connectivity is two. In fact, these two topologies are so similar that the same set of templates can be used. Some of the *templates* created for a pipeline (i.e. the templates for when a_1 or a_2 are at the end of the pipeline) would, if used, break the ring topology, but as no agent in a ring topology can have one connection, these templates are never used.

8.4 Hierarchies

In a hierarchy, each agent has two types of connections: a single connection to its *direct superior* (or, more simply, its *superior*); and potentially multiple (or zero) connections to its *direct subordinates* (or *subordinates*). There is always one, and only one, exception to this: the *root* of the hierarchy is a single agent that has no *superior*. Additional constraints can be added, such as that there exist no loops in a hierarchy, but these are not required, since the above constraints fully define a hierarchy.

An agent can have any number of *indirect superiors* and *indirect subordinates*, where an *indirect superior* is the *superior* of an agent's *direct superior* and every subsequent *superior* until the root of the hierarchy is reached. Similarly, an *indirect subordinate* is the *subordinate* of every *direct subordinate*, and every subsequent *subordinate*, until the bottom of the hierarchy is reached. To specify a hierarchy formally, we introduce the predicate $superior(a_1, a_2)$, which is true when a_1 is the *direct superior* of a_2. A multiagent system, *mas*, is thus a hierarchy if all agents except a_1 have one *direct superior* and no other, and agent a_2 has no *superior*.

$$
\begin{aligned}
hierarchy(mas) \implies\ & mas = (\mathrm{A}, \mathrm{C}) \wedge \\
& \exists a_1 \in \mathrm{A}\ \forall a_2 \in (\mathrm{A} \setminus \{a_1\}) \\
& \neg\exists a_3 \in \mathrm{A}\ superior(a_3, a_1) \wedge \\
& \exists a_3 \in \mathrm{A}\ \forall a_4 \in (\mathrm{A} \setminus \{a_3\}) \\
& superior(a_3, a_2) \wedge \neg superior(a_4, a_2)
\end{aligned}
$$

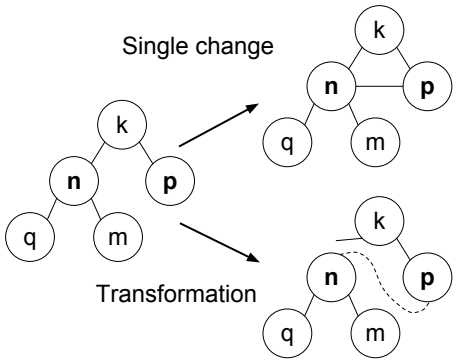


FIGURE 8.5: An example of enacting a change in a hierarchy.

The overall number of connections in a tree structure cannot be changed: for a set of agents A in a hierarchy, there are always $|\text{A}|-1$ connections. However, an agent's number of subordinates can increase or decrease, so that when one agent gains a subordinate another loses one. In this way, an organisation's structure can vary greatly while still adhering to the relevant constraints. However, the removal of a single connection always breaks a hierarchy, and an additional single connection cannot be created, because by doing so a loop always results. For example, for the hierarchy on the left in Figure 8.5, if we enact a single change, $(n, p, \text{`}create\text{'})$, then a loop is created between agents n, p, and k, as shown at the top right of the figure, breaking the topology.

8.4.1 Hierarchical Transformations

By enacting a single change on a hierarchy, the structure's topology will be broken. To ensure that this does not happen, we must instead enact *transformations*. If we consider a hierarchy to be a tree, then the basic transformation for any hierarchy is to disconnect a branch of the tree, and then reattach it, or reconnect it, elsewhere. As an example, if we take the hierarchy on the left of Figure 8.5, disconnect agent n from its superior, agent k, and reconnect it to the hierarchy by connecting it to agent p as its new superior, then we have transformed the structure while preserving the hierarchy, as can be seen at the bottom right of the figure. This *transformation* is formally represented as follows.

$$\text{CS} = \{(n, k, \text{`}remove\text{'}), (n, p, \text{`}create\text{'})\}$$

The transformation is both *legal*, and *minimal*. In addition, it contains our original change, $(n, p, \text{`}create\text{'})$, and so by using this *transformation* we can perform the change

shown at the top right of Figure 8.5, with an additional change, and topology is preserved.

8.4.2 Generating Transformation Templates

The transformation described above is only one of many possible transformations. We could have instead disconnected agent p from its superior, and then connected p to n. There are also cases in which this *transformation* cannot be used, depending on where these agents are in relation to each other. For example, if agent p is a direct superior of agent n, then disconnecting p from its current superior, and then connecting it to n as a subordinate will create a loop. More generally, to connect an agent a_1 to an agent a_2, both agents must first be *disconnected* so that there is no direct or indirect connection between them. They must then be reconnected with a direct connection. Both steps are discussed below.

Disconnect agents The most general case is if neither a_1 nor a_2 is a *direct* or *indirect superior* of the other. In this case, disconnection can be achieved by either agent a_1 or a_2 removing the connection with its superior. However, if a_1 is the *indirect superior* of agent a_2, then a_1 and a_2 can only be disconnected by removing agent a_2's connection to its superior. This case, itself, has an exception if a_1 is also the root of the hierarchy. In this case, the two agents can be disconnected by either removing the connection between a_1 and the subordinate that indirectly connects it to a_2, or removing a_2's superior.

Reconnect agents Now that agents a_1 and a_2 have been disconnected from each other, we need to reconnect them with a direct connection. This is a simple process, requiring a connection to be created between agents a_1 and a_2, such that the agent that has no *superior*, is the new *subordinate*.

From the description above, it is clear that the *disconnection* of a_1 and a_2 is dependent on their positions in the hierarchy, and their positions in relation to each other. To demonstrate, we assume a hierarchy with agents k, m, n, p, and q that are connected as illustrated on the left of Figure 8.6, and show how a connection is created between agents p and n.

When adapting a hierarchy, the most general *transformation* is to disconnect an agent from its current *superior*, and connect it to another agent, as its new *superior*. For example, Figure 8.6 shows the *transformation* being used in two ways to create a connection between agents n and p: the top instance disconnects agent n from its current *superior*, and creates a connection between n and p, making p its new *superior*. Alternatively, the bottom instance disconnects p from its current *superior*, and creates a connection between p and n, making n its new superior.

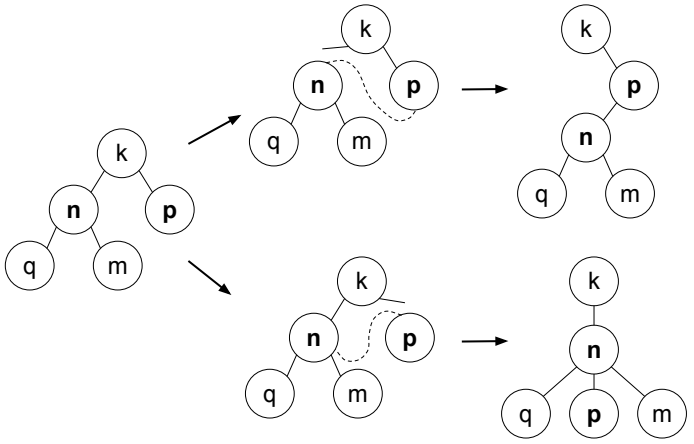


FIGURE 8.6: Hierarchy transformations when agents n and p are not in the same sub-tree, and neither is root.

FIGURE 8.7: Hierarchy transformations when both agents n and p are in the same sub-tree, but neither is root.

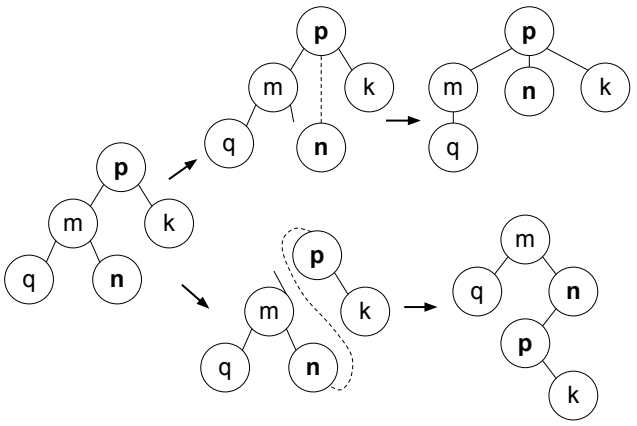


FIGURE 8.8: Minimal hierarchy transformations when both agents n and p are in the same sub-tree, and p is root.

As Figure 8.6 shows, both instances of the *transformation* maintain the hierarchy. However, under particular conditions we must restrict this. In Figure 8.7 we can see that when agent p is an indirect superior of n, the topology is not always maintained. The top of Figure 8.7 shows that by disconnecting agent n from its current *superior*, and connecting it to agent p as its new *superior*, agents n and p can be connected while maintaining the hierarchy. However, the bottom of the figure shows that by disconnecting agent p from its current *superior*, and connecting it to agent n as its new superior, the hierarchy is broken. More specifically, the topology is broken in two ways: first, p's *superior* is disconnected from all other agents; and second, a loop is created between agents p, n, and m.

This is because removing the connection between an agent and its superior must achieve two things. First, it must enable an agent to connect to a new *superior* by removing its current *superior* and, second, it must break any indirect connection between an agent and its potential new *superior*. By examining the bottom transformation in Figure 8.7, we can see that after removing agent p's current superior, it is still connected to agent n, though indirectly. Therefore, if one of the agents is a *direct* or *indirect superior* of the other, then the *transformation* can only disconnect and reconnect the agent that is the *direct* or *indirect subordinate*. Put simply, only the lowest agent in the hierarchy can be disconnected and reconnected.

One final special case arises when either agent n or agent p is the *root* of the hierarchy, such as agent p in the hierarchy seen on the left hand side of Figure 8.8. Because agent p is the root, it is the *indirect superior* of every other agent, so agent n can be

Conditions		**Transformation templates**
a_2 or a_1 indirect superior	a_1 or a_2 is *root*	
F	F	$\{(a_1, a_3, \text{'remove'}), (a_1, a_2, \text{'create'})\}$ Precondition: $superior(a_3, a_1)$ Postcondition: $superior(a_2, a_1)$
T	F	$\{(a_1, a_3, \text{'remove'}), (a_1, a_2, \text{'create'})\}$ Precondition: $superior(a_3, a_1) \wedge a_1$ below a_2 in hierarchy Postcondition: $superior(a_2, a_1)$
T	T	$\{(a_1, a_3, \text{'remove'}), (a_1, a_2, \text{'create'})\}$ Precondition: $superior(a_3, a_1) \wedge a_1$ below a_2 in hierarchy Postcondition: $superior(a_2, a_1)$ or $\{(a_1, a_3, \text{'remove'}), (a_1, a_2, \text{'create'})\}$ Precondition: a_1 is root $\wedge$ a_3 on path to a_2 Postcondition: $superior(a_2, a_1)$

TABLE 8.2: All minimal change sets containing a change to connect agents a_1 and a_2 in a hierarchy

disconnected from its current *superior*, and connected to agent p as its new *superior* (top *transformation* in Figure 8.8), but it is not possible to disconnect p from its superior, and connect it to agent n.

However, because agent p is now the *root* of the hierarchy, we have an additional *transformation* available to us. Once, again, disconnecting an agent from its current *superior* must achieve two things: enable the agent to connect to a new *superior*, by disconnecting from its old *superior*; and remove any indirect connection between an agent and its potential new *superior*. Now, because agent p is the *root* of the hierarchy, it has no superior, so the *transformation* need only remove the indirect connection between agent p and n. We can achieve this by removing the connection between agent p and the *direct subordinate* that *indirectly connects* p to n. This *transformation* is shown at the bottom of Figure 8.8, where p disconnects from its *subordinate*, agent m, and connects to agent n, such that n is p's new superior.

8.4.3 Transformation Templates

We can generate *transformation templates* in the same fashion as for pipelines. Each describes how a hierarchy can be changed while preserving the topology, according to

the conditions of the agents involved. These templates are shown in Table 8.2, with respect to two agents a_1 and a_2, and categorised according to whether a_1 or a_2 is the indirect superior of the other, and whether either agent is the root of the hierarchy. Given these *transformation templates*, we can take a single change that is not *legal*, and we can generate a *legal* change set that contains this change, by instantiating an appropriate *template*. This is achieved in the same fashion as with pipelines, described in Section 8.3.3.

Now, as stated previously, a *star* network is a specialised hierarchy where the depth is always two. To use the *transformation templates* we have just created, we would need to add some further constraints, but these would be so severe as to prevent use of the templates entirely. Therefore, the only transformation that can be used for a *star* network is the generic *transformation* described below.

8.5 Generic Topologies and Other Topologies

Though we have given many *transformation templates* above, each is only applicable to a specific topology. However, there is one *transformation* that can be applied to any structure. As well as *hierarchies* and *pipelines*, we have also used *random structures* and *fully connected structures*, yet we have not offered *transformation templates* for either. We discuss each of these issues in what follows.

8.5.1 Simple Transformations

So far *transformation templates* have been specified for each topology individually. The only *transformation* that can maintain all topologies is to swap the position of one agent with another, such as swapping p with k, by essentially disconnecting p and k from all of their current neighbours, and then creating a connection between p and each of k's previous neighbours, and creating a connection between k and all of p's previous neighbours. While this simple transformation can be used on structures constrained by any topology, specifying a *transformation template* for it is more difficult. For the hierarchy in Figure 8.9(c), to connect p to m while maintaining the topology, a connection can be created between p and m, and the connection between p and n can be removed. However, if we apply the same transformation to the pipeline in Figure 8.9(a) or the ring in Figure 8.9(b), then each respective structure is broken.

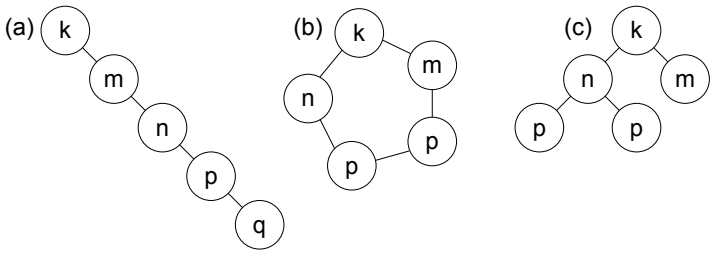


FIGURE 8.9: Five vertices connected in a line, ring, and tree structure respectively.

This ensures that structural constraints are preserved, regardless of the structure's topology, but to do so, many additional changes are needed. To move p next to m, p must disconnect from all of its previous neighbours. For a pipeline this is little different from any *transformation* generated from a pipeline's specific set of *templates*, but for a hierarchy this can be the difference between changing two connections and changing *all* of an agent's connections. In addition, to move p next to m, we can swap the position of m and k, but in the process k is disconnected from all of its neighbours when it was previously irrelevant to the *transformation*. This *transformation* thus requires a large number of changes for a potentially small improvement.

8.5.2 Unconstrained and Fully Constrained Structures

Some structures do not require the use of *transformation templates* to generate a *transformation*. In a random structure (a structure that is not constrained by any topology), there are no constraints to break, so all changes in the *initial change set* are legal, and there is no need for it to be altered. There is also no need to use the templates that are applicable to all structures, described above, but this is not to say that they cannot be used. Conversely, fully connected structures also need not use templates since they are sufficiently constrained that no changes are possible, making the use of *transformations* and *templates* redundant.

8.6 Generating Final Change Sets

To recap, in this chapter we have so far shown how a single change can break a structure's topological constraints. However, by ensuring that a single change is part of a

transformation, we can ensure that the change is enacted, while preserving topology. Finally, by using *transformation templates* given in Tables 8.1 and 8.2 we can easily generate *transformations* for *pipelines* and *hierarchies*, respectively, by instantiating a *transformation template* with the agents from a single change.

Now, as stated previously, our aim is to use *transformation templates* to ensure that each change in an *initial change set* can be combined into a *final change set*, that consists of as many initial changes as possible, while ensuring that topology is preserved. However, we have only shown how this can be achieved for a single change. In this section, we show how this can be achieved for multiple changes.

8.6.1 Union of Change Sets

The most simple, and perhaps naive, way to generate a *final change set* is to generate a *transformation* for each element of an *initial change set*, and then create the union of all these transformations, but by doing so conflicts are created. To demonstrate, suppose we have a pipeline as shown in Figure 8.10(a), and the initial change set $\text{ICS} = \{(m, p, \textit{'create'}), (m, q, \textit{'create'})\}$. For each of these changes, if we instantiate an appropriate *transformation template*, then we can generate the following two *transformations*.

$$\text{CS}_1 = \{(m, k, \textit{'remove'}), (p, n, \textit{'remove'}), (k, n, \textit{'create'}), (m, p, \textit{'create'})\}$$
$$\text{CS}_2 = \{(m, k, \textit{'remove'}), (m, n, \textit{'remove'}), (k, n, \textit{'create'}), (m, q, \textit{'create'})\}$$

If, in seeking to combine these changes into one larger set, we create their union, $\text{FCS} = \text{CS}_1 \cup \text{CS}_2$), then some of the changes conflict, as shown in Figure 8.10(b) where each cross represents the removal of a connection, and each dotted line represents the creation of a

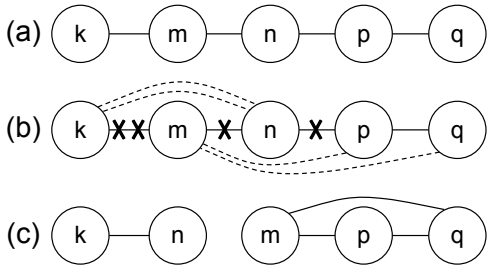


FIGURE 8.10: Applying changes from two change sets.

connection. Here, some connections are removed twice (denoted by a double cross), such as between k and m, and some are created twice. Even if we ignore duplicate changes, we can see in Figure 8.10(c) that the resulting structure is no longer a pipeline. This is caused by *transformations* being generated and enacted concurrently.

8.6.2 Preserve Topology

To ensure that such conflicts do not occur, we consider each change $cc \in$ ICS sequentially: for each *change*, a *transformation* is generated and added to the *final change set*, ensuring that no conflicting changes are included. In addition, each subsequent *transformation* is not generated with respect to the current organisational structure, but rather with respect to an internal model of the structure after all current changes in the *final change set* have been enacted, ensuring that all additional changes are generated using up-to-date information. This internal model is called the *organisational structure model*, *osm*, and the function $updateModel(osm, \text{CS})$ is used to update the current model according to the set of changes CS.

Algorithm 6 *preserveTopology*(ICS,*mas*)

1. CS $\leftarrow \emptyset$
2. FCS $\leftarrow \emptyset$
3. SCS $\leftarrow \emptyset$
4. $osm \leftarrow buildModel(mas)$
5. **for** $cc_{ics} \in$ ICS **do**
6. **if** $cc_{ics} \in$ FCS **then**
7. SCS $\leftarrow$ SCS $\cup \{cc_{ics}\}$
8. **else**
9. CS $\leftarrow generateTransformation(cc_{ics}, osm)$
10. $confict \leftarrow false$
11. **for** $(a_1, a_2, act) \in$ CS **do**
12. **if** $[act =$ *'create'*$] \wedge (a_1, a_2,$ *'remove'*$) \in$ SCS **then**
13. $conflict \leftarrow true$
14. **if** $[act =$ *'remove'*$] \wedge (a_1, a_2,$ *'create'*$) \in$ SCS **then**
15. $conflict \leftarrow true$
16. **if** $conflict = false$ **then**
17. FCS $\leftarrow$ FCS $\cup$ CS
18. $osm \leftarrow updateModel(osm, \text{CS})$
19. **for** $(a_1, a_2, act_x) \in$ FCS **do**
20. **for** $(a_1, a_2, act_y) \in$ FCS **do**
21. **if** $[act_x =$ *'create'*$\wedge act_y =$ *'remove'*$] \vee [act_x =$ *'remove'*$\wedge act_y =$ *'create'*$]$ **then**
22. FCS $\leftarrow$ FCS $\setminus \{(a_1, a_2, act_x), (b_1, b_2, act_y)\}$
23. SCS $\leftarrow$ SCS $\cup \{cc_{ics}\}$

By considering each change in the *initial change set* sequentially, we ensure that each *transformation* is atomic, removing any conflicts between multiple *transformations*. This approach is specified formally in Algorithm 6, and discussed below. As parameters, the algorithm expects an *initial change set*, ICS, and a multiagent system *mas*. It begins by creating an empty *change set*, CS, *final change set*, FCS, and *successful change set*, SCS. The set SCS is used to track all changes from the *initial change set*, that have been successfully added to the *final change set*. The internal model of the organisational structure is also initialised to the current structure of *mas* on line 4, using the function $buildModel(mas)$.

For every change $cc \in$ ICS, we begin by checking if *cc* already exists in the *final change set* on line 6. If it does, then it has already been added, so no *transformation* is needed. Instead, the change is just added to SCS on line 7 and the next change is considered. If it does not exist, then a *transformation*, CS, is generated on line 9, using the function $generateTransformation(cc, osm)$, which uses *transformation templates* to produce a change set that contains the change *cc*, and any additional changes according to the *template* used. It is important to note that the *transformation*, CS, is generated in the context of *osm*, rather than the current structure of the *multiagent system*, ensuring that the *transformation* is generated with the most up-to-date information. Next, on lines 10–15, each change in the generated *transformation* is checked, to ensure that it does not reverse any changes that have already been added to SCS. If a conflict is found, then the entire *transformation* is discarded. Note that changes in CS can reverse changes that are not in SCS, but are in FCS.

If no conflicts are found between SCS and CS, then CS is added to the *final change set* on line 17, and *osm* is updated on line 18. Finally, the *final change set* is checked for any contradicting changes. If two changes exist such as $(a_1, a_2,$ *'create'*$)$ and $(a_1, a_2,$ *'remove'*$)$ that state two opposing operations, then they cancel each other out, so both are removed (lines 19–22). At this point we have a *final change set* that contains the change *cc* and is *legal*, so on line 23 *cc* can be added to SCS, before continuing to the next change in ICS.

8.7 Experiments and Results

In this chapter we have shown how structure can be adapted while preserving a particular topology. Our aim in this evaluation is to: first, evaluate whether structural adaptation

can improve the performance of topologically constrained organisations, when compared to static structures; and second, determine whether there is any increase, or decrease, in performance, when compared to structures that are not topologically constrained.

We ran similar experiments to previous ones, using the experimental setup described in Chapter 4 and the experimental parameters described in Section 4.4. However, we also introduced some minor differences. In Chapter 7, the use of random structures and unconstrained reorganisation made it difficult to detect network disconnection, which ultimately caused tasks to fail, and also caused *network based search* to continue indefinitely. For this reason, the search depth of the *network based search* approach was limited. However, one of the benefits of maintaining a topology is that disconnection is no longer an issue, so *network base search* no longer has a depth limit.

We have already shown that *topology preserving reorganisation* is made up of two functions: *suggestStructuralChanges* and *preserveTopology*. The function *suggestStructuralChanges* is instantiated by either *random reorganisation*, *full service connection*, or *frequent service connection* from Chapter 7, and the function *preserveTopology* has been the main focus of this chapter, where either *pipelines* or *hierarchies* are preserved. First pipelines are evaluated, with and without reorganisation, and against random structures. This is followed by the same evaluations for hierarchies.

As with all previous experiments, results are plotted as averages over time. Where appropriate we have endeavoured to include error bars to show the distribution of the results. However for brevity and ease of understanding we have, in some cases, chosen to merge the results from multiple experiments into a single summary graph and excluded error bars. In such cases, the fully plotted results, along with error bars, can be seen in Appendix B.

8.7.1 Pipelines

In order to evaluate reorganisation while maintaining *pipeline* topologies, we use the *transformation templates* specified at the end of Section 8.3, in conjunction with random reorganisation, full service connection, and frequent service connection. Recall, in Chapter 6 we estimated that service location time for a static pipeline of 100 agents is between 1300 and 2550 time steps, depending on the position of the agent in the pipeline. In addition, many services failed to be located. We expect *service location time* to be significantly decreased when reorganisation is used.

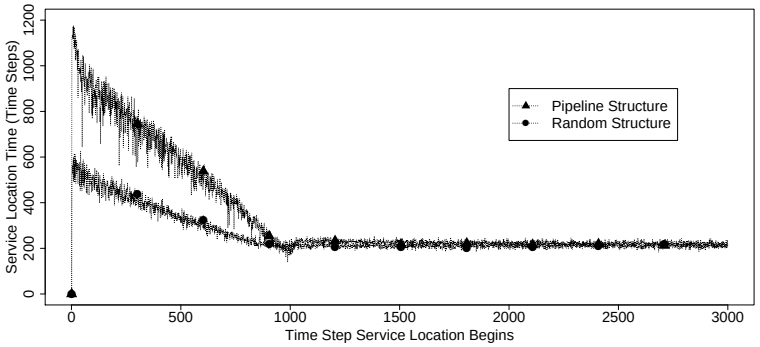


FIGURE 8.11: Random vs. pipeline structure, using random reorganisation: service location time

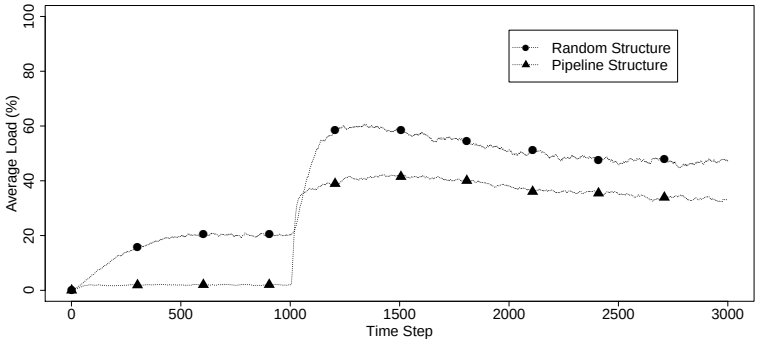


FIGURE 8.12: Random vs. pipeline structure, using random reorganisation: average load

Random Reorganisation By using random reorganisation, *service location time* is decreased to approximately 200 time steps as shown in Figure 8.11. This is also the same performance achieved when a random structure is used (and so no topology is being maintained). In addition, as can be seen between time steps 1 and 999 in Figure 8.12, when a pipeline structure is static, average load is approximately 2% because no tasks are being allocated, due to poor *service location time*. However, at time step 1000 reorganisation begins, and so load increases rapidly. A pipeline that is randomly reorganised does not achieve the same average load as a random structure, but there is a slight decrease in load after approximately 2000 time steps as the backlog of tasks is executed. Because tasks wait longer to be executed (Figure 8.13), average load is less than when there is no defined topology (Figure 8.12), though in both cases all tasks are

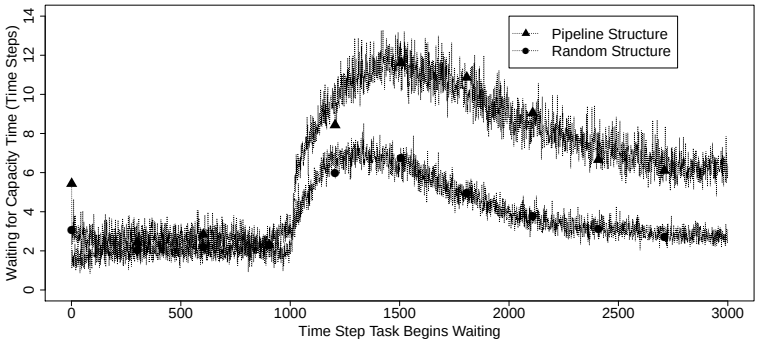


FIGURE 8.13: Random vs. pipeline structure, using random reorganisation: waiting for capacity time.

executed. Finally, *waiting for capacity time* drops to between six and seven time steps when the backlog of tasks is completed.

Full Service Connection When *full service connection* reorganisation is used there is an initial improvement in overall performance, when compared to static pipelines, but this improvement is slowly reversed after the initial increase. In Figure 8.14 we can see that *service location time* initially decreases, but it is much higher than when *full service connection* is used on a random structure and, subsequently, begins to increase again. This can be more clearly seen in Figure 8.15 where load increases, from approximately 2% to over 40% within a few time steps of reorganisation starting, but steadily declines over the remainder of the simulation. Similarly, in Figure 8.16, the waiting time for a task increases as more tasks are successfully allocated, decreasing later, and with higher variance.

This is caused by the restrictive nature of *pipelines*. When there are no topological constraints, *frequent service connection* increases the number of connections until an agent has at least one connection to an agent offering each service. However, when maintaining a *pipeline*, each agent can only have a maximum of two connections, so if an agent already has two connections, then any further changes will remove previously created connections. Since agents use some services more than others, it is likely that the first connections created will be to other agents offering the most regularly used services, but these are also likely to be the first connections to be removed when further reorganisation reverses these changes. Thus, by the end of the simulation, agents have

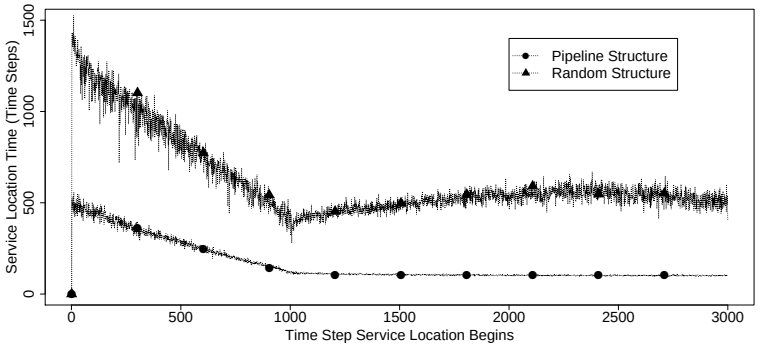


FIGURE 8.14: Random vs. pipeline structure, using full service connection: service location time.

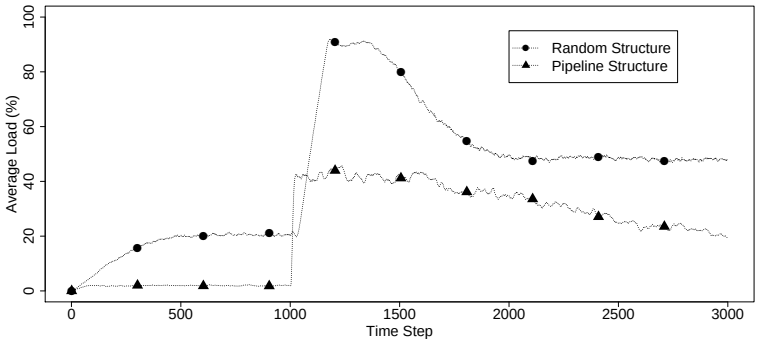


FIGURE 8.15: Random vs. pipeline structure, using full service connection: average load

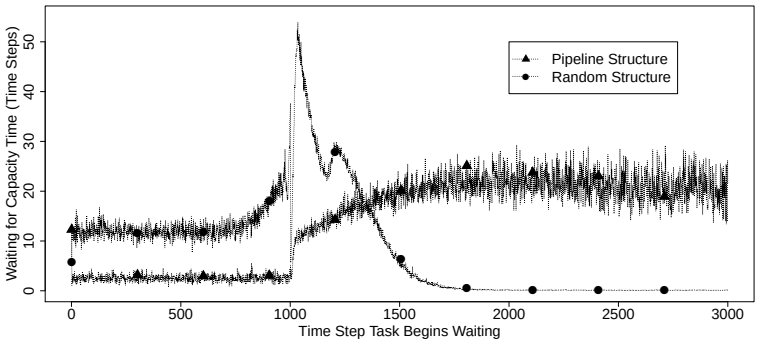


FIGURE 8.16: Random vs. pipeline structure, using full service connection: waiting for capacity time

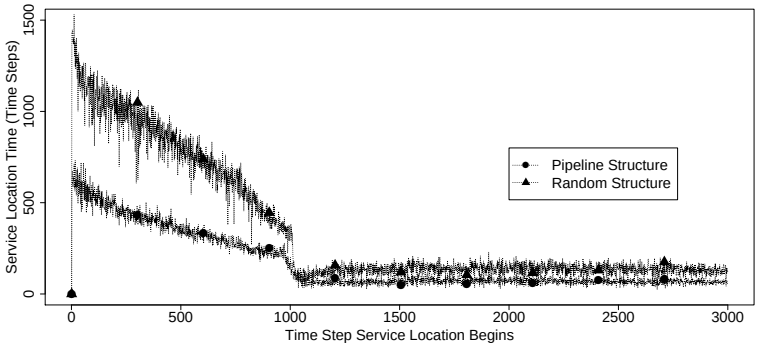


FIGURE 8.17: Random vs. pipeline structure, using frequent service connection: service location time

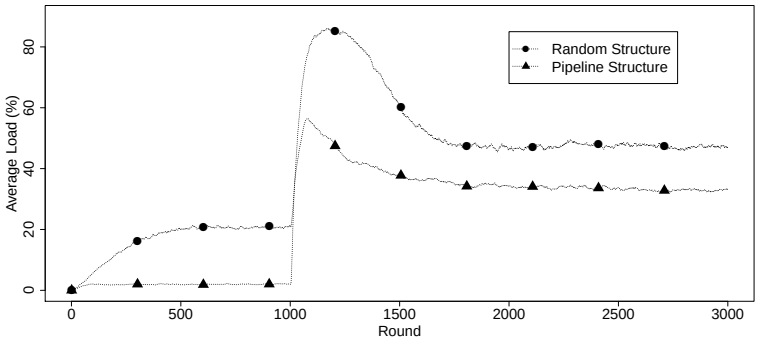


FIGURE 8.18: Random vs. pipeline structure, using frequent service connection: average load

mostly disconnected from others offering their most regularly used services, and are instead connected to those offering services used only occasionally.

Such a phenomenon could be addressed by adapting *full service connection* so that an agent periodically checks the services its current neighbours offer, and create connections to agents offering the missing services. However, this does not overcome the limitation of two connections. For example, if an agent requires 10 services, but can only connect to two other agents, then it cannot connect to enough agents to ensure that each required service is offered by a neighbour. Instead, it must prioritise services, in a similar fashion to *frequent service connection*.

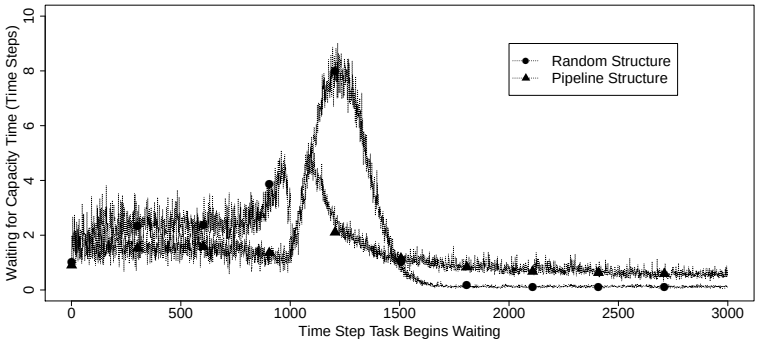


FIGURE 8.19: Random vs. pipeline structure, using frequent service connection: waiting for capacity time

Frequent Service Connection When *frequent service connection* reorganisation is applied to pipeline structures, performance is significantly better than reorganising pipelines with *random* reorganisation and *full service connection* reorganisation (with p-values of 1.37×10^{-25} and 1.44×10^{-57} respectively). In Figure 8.17 we can see that *service location time* varies between around 100 and 200 time steps, which is an improvement over *random* reorganisation, where service location takes just over 200 time steps (Figure 8.11), and this performance is sustained, unlike when *full service connection* is used (Figure 8.14). However, *service location time* is still higher than when *frequent service connection* is used on a random structure, which is also shown in Figure 8.17. In Figure 8.18, the average load increases almost instantly when reorganisation starts, then begins to decrease, and finally reaches a plateau, similar to when *random* reorganisation is used on a pipeline (Figure 8.13), but with sharper changes. The same sharp increase, and decrease, can be seen for waiting for capacity time in Figure 8.19.

As with *frequent service connection* reorgnisation in the previous chapter, we want to explore whether these results only occur with the specific parameters used in this experiment. To test this, we repeated this experiment with 25, 50, and 75 agents. Figure 8.20 shows that as the number of agents decreases, average service location time also decreases, and this is expected — the fewer agents there are, the easier it is to locate services. At time step 1000, when reorganisation begins, we can see that regardless of the number of agents, the same pattern of improved service location time is seen. We also repeated the experiment for 50, 100, 150, and 200 services, and Figure 8.21 shows that regardless of the number of services, the same pattern of improved service location time can be seen when reorganisation begins.

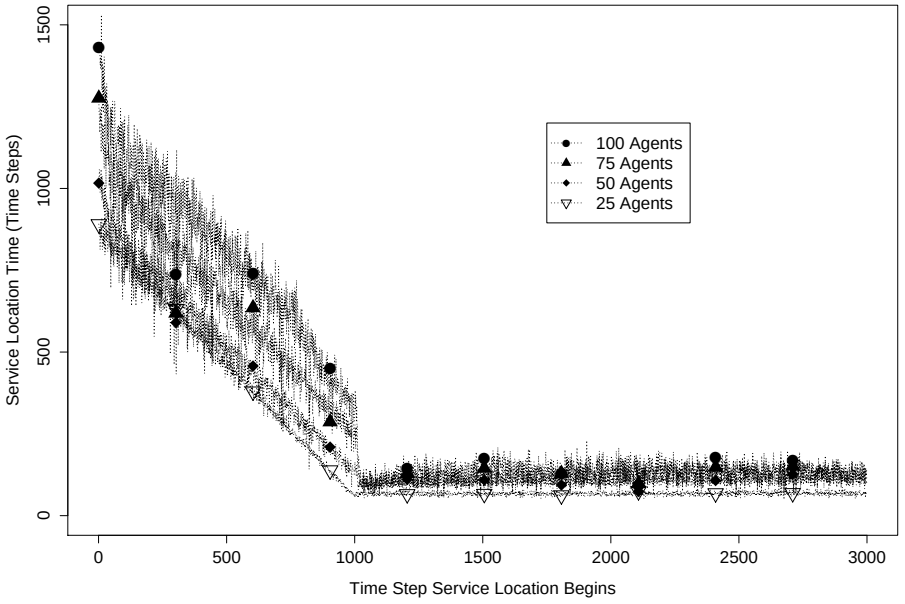


FIGURE 8.20: Average Service Location Time: Pipeline - Frequent Service Connection, varied number of agents

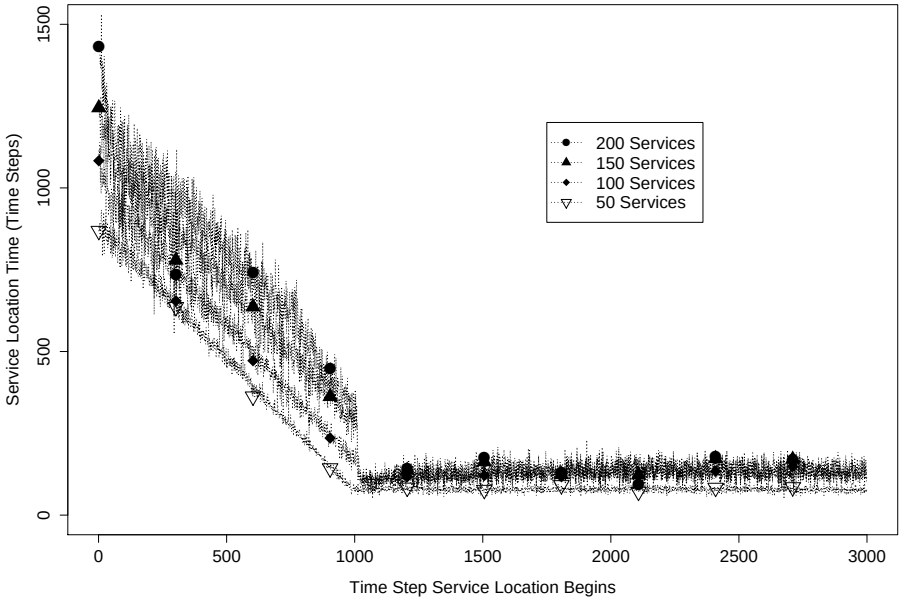


FIGURE 8.21: Average Service Location Time: Pipeline - Frequent Service Connection, varied number of services

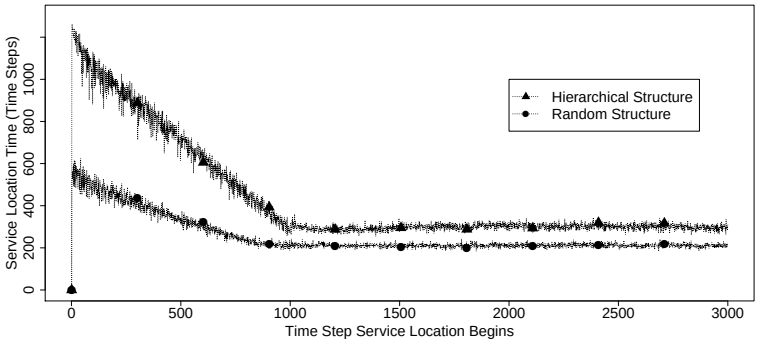


FIGURE 8.22: Random vs. hierarchical structure, using random reorganisation: service location time

Summary Through experimentation we have compared the performance of static pipelines, and pipelines and random structures that are reorganised by *random reorganisation*, *full service connection*, and *frequent service connection*. We have shown that, in general, any form of reorganisation can improve the performance of a pipeline, when compared to a static structure. *Full service connection* improves performance, though this improvement is not sustained. However, *random reorganisation* and *frequent service connection* both achieve significant improvements for *service location time*. In addition, when compared to random structures (structures that do not preserve any topology), we found that by preserving pipelines only a slight decrease in performance is observed.

8.7.2 Hierarchies

When considering *hierarchies*, we use the *transformation templates* introduced at the end of Section 8.4, in conjunction with random reorganisation, full service connection, and frequent service connection. Recall that in Chapter 6 we estimated that for the static hierarchies generated in our experiments, *service location time* is between 1,074 and 3,258 time steps. As with pipelines, we expect that reorganisation will improve *service location time*.

Random Reorganisation When *random* reorganisation is used on a hierarchy, performance as a whole is improved when compared to static hierarchies: Figure 8.22 shows that once reorganisation begins, average *service location time* decreases to approximately

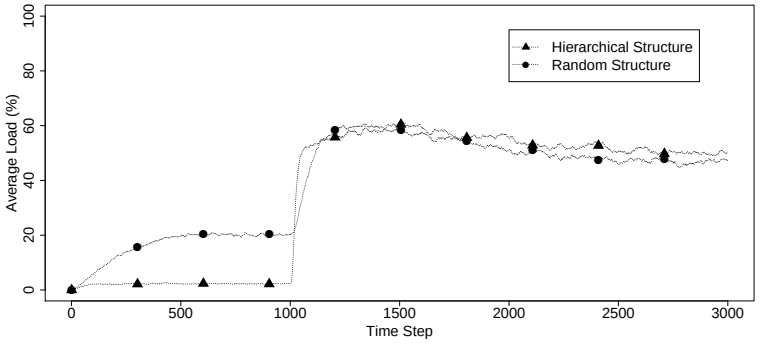


FIGURE 8.23: Random vs. hierarchical structure, using random reorganisation: agent load

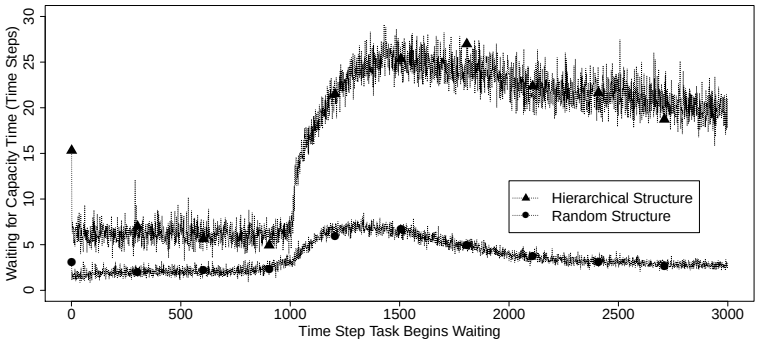


FIGURE 8.24: Random vs. hierarchical structure, using random reorganisation: waiting for capacity time

300 time steps. Tasks are allocated at a faster rate, causing the average load to increase, initially rapidly, and within around 500 time steps to as high as 60% (Figure 8.23). As the backlog of tasks is completed, load decreases to 50% after approximately 1500 time steps, as with random reorganisation on a random structure. Because more tasks are successfully allocated after reorganisation begins, there is competition over agent capacity to execute tasks, so *task waiting for capacity time* increases quickly (Figure 8.24).

Full Service Connection When hierarchies are reorganised using *full service connection*, overall performance is improved when compared to static hierarchies and, as with pipelines, this effect is slowly reversed by further reorganisation. Figure 8.25 shows the initial improvement in *service location time*, which drops to just below 500 time steps,

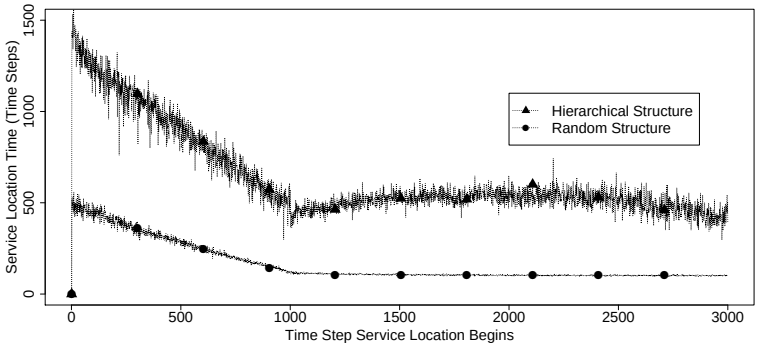


FIGURE 8.25: Random vs. hierarchical structure, using full service connection: service location time

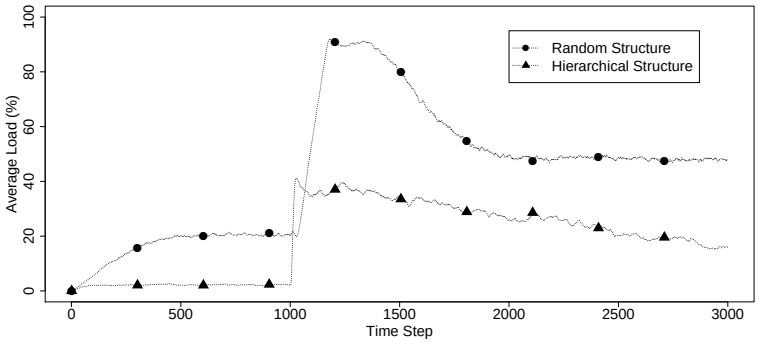


FIGURE 8.26: Random vs. hierarchical structure, using full service connection: average load

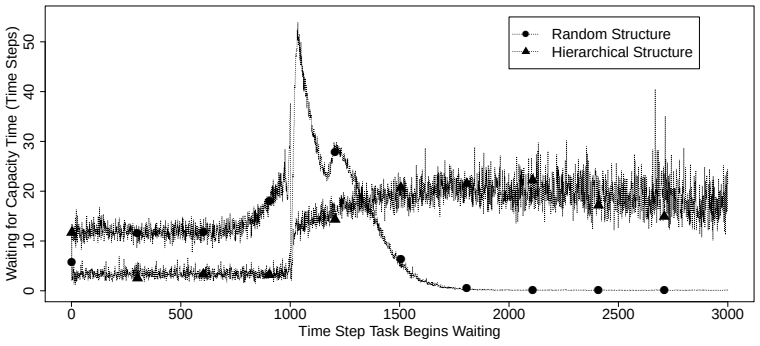


FIGURE 8.27: Random vs. hierarchical structure, using full service connection: waiting for capacity time

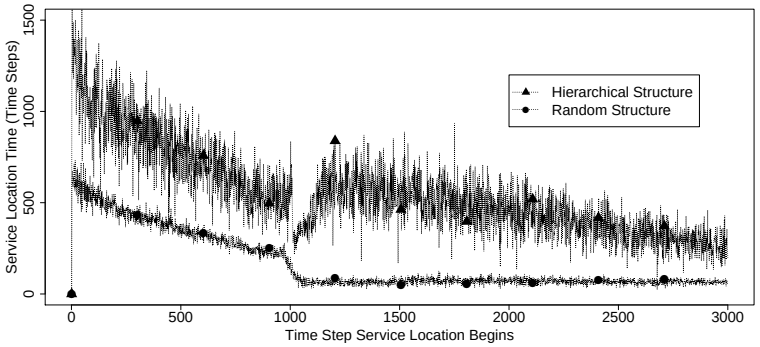


FIGURE 8.28: Random vs. hierarchical structure, using frequent service connection: service location time

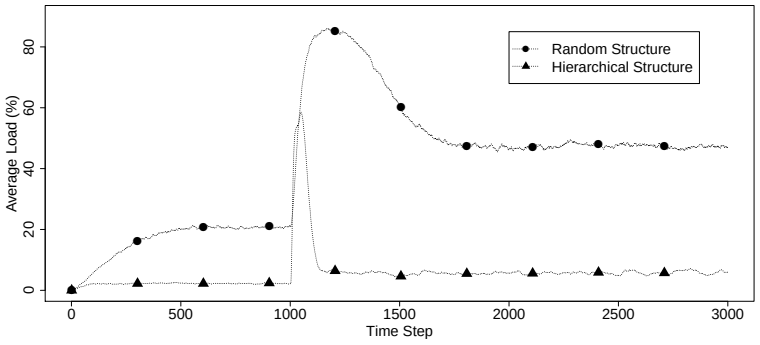


FIGURE 8.29: Random vs. hierarchical structure, using frequent service connection: average load

before increasing slightly. We can see in Figure 8.26 that *full service connection* causes load to increase rapidly to around 40%, before slowly decreasing to below 20% by the end of the simulation. The same trend can be seen for *waiting for capacity time*; as tasks take longer to be allocated, there is less competition for agent load, and so tasks wait less time to be executed (Figure 8.27).

Frequent Service Connection Interestingly, here *frequent service connection* no longer provides the best performance overall. In Figure 8.28 *service location time* initially decreases, but quickly increases again. Though service location time begins to slowly decrease approximately 200 time steps after reorganisation begins, this is not because *service location time* is improving, but rather because from approximately time

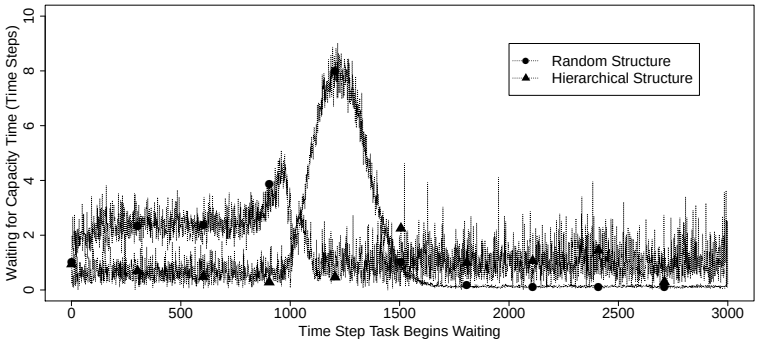


FIGURE 8.30: Random vs. hierarchical structure, using frequent service connection: waiting for capacity time

step 1200 onwards, an increasing proportion of service location processes are not completing before the end of the simulation. With the worst results not being accounted for, average *service location time* appears to decrease. This is supported by the results for load in Figure 8.29), where load initially increases to 60%, but quickly decreases again, to below 10%. The same trend can also be seen for *waiting for capacity time* (Figure 8.30).

This poor performance can be attributed to the operation of *frequent service connection* itself. Assume we have a set of agents that have been reorganising for some time, and so are close to their regularly used services, and far from rarely used services. If an agent receives an arbitrary task from the environment, then the first required service may be offered by an agent in the system. According to our estimates in Chapter 6, in the worst case a hierarchy will take 9,816 time steps to locate this service, though as reorganisation moves rarely used services further away this can increase further.

Once the service has been located, the task is allocated to the appropriate agent, and the first requirement is satisfied. Now, if the task has further requirements, according to our *requirement correlation model* in Chapter 3, all required services will be from a single *service category*, and these will tend to be an agent's most frequently used services. *Frequent service connection* has moved these services closer to each agent, and so the time needed to locate these services is likely to be low.

From this we can conclude that locating a task's first required service is slow, but locating all subsequent required services is fast. However, because the worst case for hierarchies is so high, in many cases the first required service is not located within the time of the

simulation, and so we cannot see the benefit of locating frequently used services. Indeed, because so few tasks are being executed it is unclear whether agents have yet determined what their must frequently used services are. This problem exists for pipelines, as well as hierarchies. However, when the search distance in a pipeline increases, search time increases polynomially; in a hierarchy as the search distance increases, search time increases exponentially, therefore the effect of worst case results are more prominent, highlighting this issue.

Summary As with pipelines, we found that reorganisation, in general, improves *service location time* for hierarchies, when compared to static hierarchies. However, *frequent service connection* no longer performs the best because by increasing the distance between agents and their occasionally used services, some tasks do not even begin within the time span of our experiments. Instead, the best performance is achieved by random reorganisation.

8.8 Conclusion

In this chapter we have described a novel reorganisation framework known as *topology preserving reorganisation*, which can reorganise an organisation while preserving the structure's original topology. We also provided six instantiations of *topology preserving reorganisation*: three of which use *random reorganisation*, *full service connection*, and *frequent service connection*, while preserving pipelines; and three of which preserve hierarchies, using the same structural adaptation approaches.

In the general case, we found that by reorganising an organisation's structure, overall system performance is improved when compared to static structures. However, there are exceptions to this, such as when using *frequent service connection* to reorganise a *hierarchy*. We also found that by preserving topology, performance is slightly decreased when compared to unconstrained (random) structures. This is expected, because structural constraints can forbid beneficial changes to the structure, that an unconstrained structure can enact.

In addition to these specific results above, broadly we have shown how the decision to adapt connections can be effectively separated from the complexity of maintaining a topology, which can be addressed once a decision has been made about all changes

to the structure. This has allowed us to use intuitions about how agents should be connected, and indeed any future intuitions, and apply them to a number of topologies with little effort. Finally, enabling any system to preserve topology while reorganising allows systems with structural constraints to make use of reorganisation approaches that have previously not been applicable.

Chapter 9

Conclusion

9.1 Introduction

This thesis has been concerned with the need for distributed systems to reorganise in support of more efficient operation. In particular, and unlike previous approaches to reorganisation, the thesis has considered the need to preserve existing topological constraints so that specific organisational structures, suited to the application at hand, can be maintained despite this reorganisation. The work has focussed on the particular case of distributed task allocation and execution, but the techniques and results are generally applicable and relevant.

This chapter is structured as follows. In Section 9.2 a summary of this thesis is provided, highlighting some major milestones in our work, along with a list of our major contributions. This is followed by a description of some of the limitations of our research in Section 9.3, and a discussion about some potential future avenues of work that build on this thesis. Finally the chapter concludes in Section 9.4, with some final remarks.

9.2 Summary and Contributions

9.2.1 Centralised and Decentralised Service Location

The thesis has covered key issues in relation to an investigation of task allocation and execution, notably examining and comparing several approaches. Beginning with an

analysis of the basic processes involved, it identified key aspects that can lead to a degradation of performance, and used these as the basis for developing techniques to improve this performance. More specifically, since the time needed to locate services is the most significant (and relevant) contributor to performance, this provided the focus for the subsequent analysis through both centralised and decentralised approaches.

In centralised service location, the amount of time needed to locate services is related to the load on the central registry. Because of the inherent limitations in centralised approaches, however, a move to consider decentralised approaches becomes important. In considering decentralised service location across different organisational structures, several conclusions became clear. First, as the number of connections increases, the distance between agents and the services that they require decreases, and thus the time needed to locate services decreases, but the number of connections that must be maintained is increased significantly. Second, a greater distribution of links across a system, such that some agents, or even a single agent, has significantly more connections than others, decreases the distance between agents and the services they require, thus improving service location time, but in doing so communication bottlenecks are introduced.

9.2.2 Structural Adaptation

Given this analysis of centralised and decentralised systems, and in particular to enable a system to adapt to environmental changes, we then moved to an investigation of three approaches to adapting the connections between agents at run time. Two of these approaches were motivated by the relationship between the distance between agents and services, and its effect on performance. By actively reducing the distance between agents and services through *structural adaptation*, service location time can be improved significantly, though it is clear that random changes can also improve performance.

Random reorganisation improves service location time because repeated changes to a structure enable an agent to have direct contact with more agents over a period of time, than would be possible at any fixed point in time, thus improving service location time. With *full service connection*, location time is decreased because the number of overall connections is significantly increased, which also directly improves service location time. Finally, *frequent service connection* causes only a slight increase in the number of connections, and the degree distribution does not change, yet service location time is improved significantly, on average. Importantly, by adopting the assumption of correlated

services (that subsets of services tend to be required together), service location time improves merely by decreasing the distance between agents and their most frequently used services. However, a small number of tasks fail to be executed.

9.2.3 Structural Adaptation Constrained By Topology

In contrast to other work, we also sought to develop techniques to preserve topological constraints rather than make arbitrary changes to a structure. Building on the techniques developed thus far, we provided techniques that not only make it possible to preserve topology when undertaking reorganisation, but also achieve performance benefits from doing so, when compared to static structures. In addition, by maintaining such structures, we can ensure that the network stays connected, so that no service location processes fail. Thus, by enabling structure to be adapted while preserving constraints, systems that would otherwise be static (such as telecommunications networks), can also adapt their structure.

9.2.4 Contributions

We can extract the key contributions from this summary more succinctly below.

- We have provided two fundamental approaches to improving task throughput, by reducing the distance between agents and the services they require.
- We have provided an approach to decrease the distance between agents and the services they require most frequently, based on the assumption that subsets of services are required or used together.
- We have introduced a framework, together with specific techniques, to adapt structure that is constrained by topology.

Moreover, there are several additional, secondary contributions, as follows.

- We provided an analysis of the process of completing tasks, resulting in a modular decomposition of task allocation and execution, and an analysis highlighting the phases of task completion that can be affected by structural adaptation.

- We revealed a correlation between the number of connections in an organisation, the variation between each agent's number of connections, and the distance between agents and services, which ultimately affects service location time.

9.3 Limitations and Future Work

Though we have sought to provide a general analysis and techniques so that our work is applicable to as wide a scope as possible, inevitably there are limitations. This section provides a discussion of these limitations and develops considerations of possible future work that builds on the work described in this thesis.

9.3.1 Simulation Environment

One of the major limitations our work is in the simulation environment itself. Ideally, we would be able to perform experiments without a limit on *network based search*, so that our results were not affected. In many cases the results were difficult to interpret because a large number of services had not been located when the experiments ended. Ideally, we could run simulations for millions of time steps, to get a more accurate representation about the performance of *network based search*, when performed on random structures, pipelines, and hierarchies.

9.3.2 Task Allocation and Execution Model

When modelling task allocation and execution we considered agents to be heterogeneous, but only to the extent that each agent offers different services. Apart from this, all agents behave in the same way. In particular, we have assumed that all agents execute tasks at the same rate, because it had little bearing on the main focus of this thesis. However, for any context in which execution speed varies greatly, allocating tasks to inappropriate devices can cause severe delays, so incorporating varied computational abilities could enhance the analysis and provide a more appropriate model.

Moreover, in our analysis of task allocation and execution we constructed a model of task completion time that can be divided into five phases. However this thesis primarily focused on *service location time*, so when modelling our problem we specified that: all

agents execute tasks at the same rate (as indicated above), so that one device is no more desirable than another; there is only one instance of each service, so load cannot be distributed; and the cost of communication is uniform regardless of the type of communication. Though none of these factors affects the validity of our results, they are all important with regard to the wider goal of improving task throughout in distributed systems.

One potential avenue for future work is therefore to extend our task allocation and execution model so that it more fully describes the environment. For example, by extending agents so that different devices execute tasks at varying speeds, and by introducing multiple instances of services, we could consider how structural adaptation affects load distribution. More specifically, we could consider, not just how to locate services, but also which particular service instance, out of all services instances, should execute a task.

9.3.3 Decentralising Topologically Constrained Reorganisation

While we have shown that it is possible to preserve topology while adapting an organisation's structure, our approach requires that reorganisation be enacted by a central entity that manages a system's organisational structure. However, as we have indicated, relying on a central registry represents a single point of failure. This is also true when using a centralised approach to reorganise, but with one key difference. Service location is a system-critical process, so that when the central registry fails the system's operation is compromised. Nevertheless, since our reorganisation approach aims to improve service location rather than enable it, if the reorganisation process fails, then the system can still function, albeit on a static organisational structure.

Though reorganisation is not a system-critical process, there are other problems that must be addressed. To preserve topology while reorganising we make two key assumptions. First, we assume that the entity that is reorganising the organisational structure has access to information about each agent in the system, and this information is readily available. Second, we assume that it is possible for this entity to assume control of the entire organisational structure. When developing an approach to preserve topology we did not consider the retrieval of information because it was out of scope of what we set out to achieve: develop an approach to adapt structure, while preserving topological constraints. However, depending on the context, these may not always be realistic assumptions. If the information required to reorganise is distributed across a large number

of agents, then collecting all this information is a complex task for one central entity. In addition, if agents are in different areas of authority then it may not be acceptable to have one entity controlling the whole organisational structure.

First and foremost, to adapt an organisational structure, information about the current structure is required — it is impossible to ensure that changing connections will not break structural constraints, without knowing the current form of the organisational structure.

Second, information about the agents in the structure is required to determine which adaptations must be made. For example *frequent service connection*, in a general sense, creates a connection between an agent and agents offering instances of its most frequently required services. This requires information about the services each agent uses, and how frequently they are required. Now, if information about a small number of agents is required, this can be retrieved by directly querying each agent for the information. However, if information is required from many agents or, in some cases, every agent in the system, then there is a potential need for a more efficient data collection approach. In such situations, when large amounts of data are being collated, it may also be useful to consider when information needs to be accurate and complete, and when partial information is acceptable, even if complete information is ideal. Each of our reorganisation approaches only relies on each agent's local information, such as service requirement frequency. However, if an alternative approach requires that an agent compare its number of connections to the average number of connections of all other agents then, to get an accurate average, the central entity needs to know how many connections each agent has. For a large agent population, a reasonable estimate can be determined by querying only a sample of the agent population.

9.4 Final Remarks

To conclude, we have developed approaches to improve task throughput, specifically focusing on decreasing the time required to locate services. We have explored the effect of decentralising service location, and analysed how services can be located across a number of structures. We have shown how simple changes to these structures can help to improve the location of services and, finally, we have introduced a framework to ensure that when an organisation structure is changed, any constraints on the structure are preserved.

More broadly, we have taken some significant steps to widen the scope for self-organisation techniques. In prior analyses, reorganisation typically changed any and all connections if so required, to ensure that system performance was improved. However, by doing so, it was impossible to constrain the reorganisation appropriately. By introducing an approach to preserve topology when making changes to an organisation's structure we have enabled self-organisation to be applied to systems that are constrained by their nature. By designing our framework in a modular fashion we have ensured that existing reorganising approaches can be easily adapted to preserve structure, and future approaches need not actively maintain topology. Instead, when structural constraints need to be preserved, any reorganisation approach can be used in conjunction with ours.

Appendix A

List of Terms

A.1 Naming Convention

Throughout this thesis notation is used to formally represent various key elements. This notation is governed by the following conventions. Sets are always represented by one or more small, upper case letters, that form an acronym of the set's name; for example A represents a set of agents. Elements in a set are represented by one or more italicised, lower case letters, such as an agent $a \in \text{A}$. There is one exception to the rule that elements are lower case: when we describe a set of sets, each element is represented by small upper case characters, in a similar fashion to all other sets. To help identify between multiple elements from the same set, a subscript is used to number each element. Finally, all functions are represented by full words, such that all characters are italicised, and lower case, with the exception of the first character of the second word, third word, and so on, which is upper case (this is typically referred to as camel case).

A.2 Terms

A.2.1 Sets

A The set of all agents.

C A set of agents that an agent is connected to.

CS Change set. Set of elements describing changes to an organisation's topology.

ES Set of services that an agent has, so far, encountered.

F Set of most frequently used services.

FCS A change set, known as a final change set.

ICS A change set, known as an initial change set.

LWD Set of requirements, each of which is dependent on others, but of which no others depends.

NAD Set of requirements, each of which is not a dependant of another.

R A set of requirements contained in a single task.

ROT Requirement Ordering Tree. A set of constraints between requirements.

RS Set of services that an agent has, so far, required.

RT A set of running tasks. Tasks that an agent has started to execute.

S The set of all services.

SC A service category.

SCS Successful change set.

SI A set of service instances.

T A set of tasks that an agent has received.

A.2.2 Variables

a, k, m, n, p, q, u Each refers to an agent.

lra, pa, dsu, sc Devices from the eScience scenario.

act Denotes the action taken performed by a change: remove or create.

b The number of children at each level of a search tree.

c A connection between two agents.

cc A single connection change to an organisation's topology.

cr A central registry for agents or services.

d The depth of a search tree.

dr Dependant requirement.

e The distance from an agent, to the nearest end of a pipeline.

l The length of a pipeline.

mas Multiagent system.

nc The number of connections to change when randomly reorganising.

nod A requirement's number of dependants.

nor A task's number of requirements.

ns Number of services in a particular system.

nsc Number of service categories.

nsi Number of service instances offered by a single agent.

osm A model of the organisational structure.

pr Parent requirement.

r A requirement.

rot A dependency between requirements.

rt A running task.

rtc An agent's running task capacity. The number of tasks that an agent can run at once.

s A service.

si A service instance.

srt Service required rime. The number of rounds that a service is required.

t A task.

A.2.3 Functions

$buildModel(mas)$	Returns a model of the current organisational structure.
$connectedTo(a)$	Returns the set of agents, to which agent a is connected.
$createConnection(a_1, a_2)$	Creates a connection between agent a_1 and a_2.
$degree(a)$	Function to calculate the number of connections an agent a has.
$distance(a_1, a_2)$	Function to calculate the distance between agents a_1 and a_2.
$enact(\text{FCS}, mas)$	Enacts the changes in FCS onto the structure of *mas*.
$frequentServiceConnection(a, \text{RS})$	Reorganises agent a's connections using frequent service connection.
$fullServiceConnection(a, \text{RS}, \text{ES})$	Reorganises agent a's connections using full service connection.
$G(\text{N}, p)$	Generates a random network according to Erdős and Renýi's model for random graphs.
$generateTransformation(cc, osm)$	Generates a transformation, in the context of *osm*, that contains *cc* .
$generateTree(\text{R})$	Generates a requirement ordering tree, including all requirements in R.
$networkBasedSearch(a, s)$	Performs network based search from agent a, for service s.
$offeredBy(a)$	Returns the set of services offered by agent a.

$pipeline(mas)$	Returns true if *mas* is a pipeline.
$preserveTopology(\text{ICS}, mas)$	Filters ICS to ensure that it will preserve topology.
$preserveTopologyReorganisation(mas)$	Reorganises agent a's connections, while ensuring topology is preserved.
$rand(\text{A})$	Returns a random element from the set A.
$random(x, y)$	Returns a random value between x and y.
$randomReorganisation(\text{A})$	Randomly reorganises a set if agents, A.
$removeConnection(a_1, a_2)$	Removes a connection between agent a_1 and a_2.
$searchTime(b, d)$	Returns the time for network based search to search d levels, when each node expands to b children.
$sortByFrequency(\text{S})$	Sorts the set of services in set S by how frequently they have been required.
$suggestStructuralChanges(mas)$	suggests changes to an organisational structure.
$superior(a_1, a_2)$	Returns true if a_1 is the superior of a_2 in a hierarchy.
$top(\text{RS})$	Returns the set of top required services.

Appendix B

Results With Error Bars

B.1 Introduction

Throughout this thesis, summary graphs are often provided for brevity and ease of understanding. However, for completeness, this section provides plots to show average service location time, with error bars to confirm the accuracy of these results (+/− one standard error).

B.2 Service Location Time When Adapting Structure

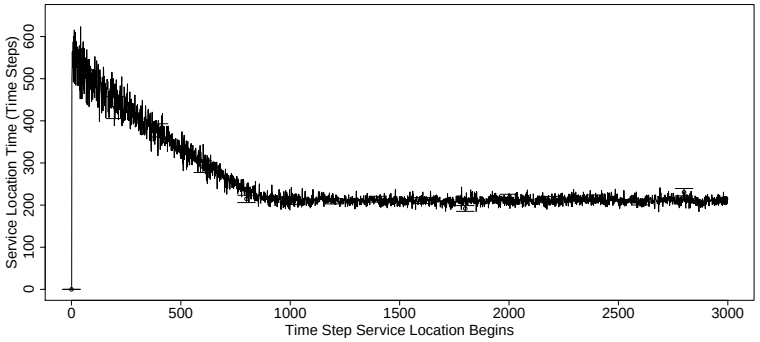


FIGURE B.1: Service location time, random reorganisation

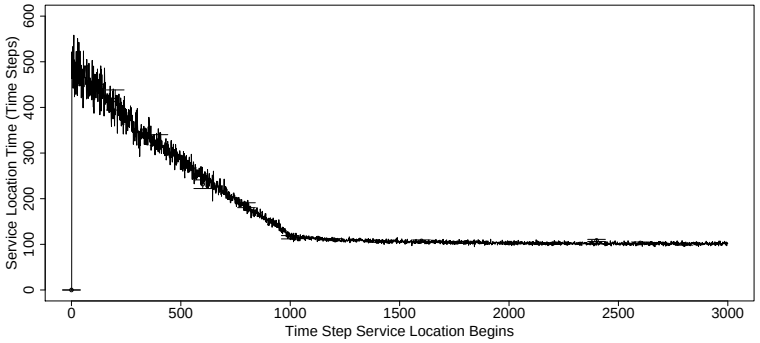


FIGURE B.2: Service location time, full service connection reorganisation

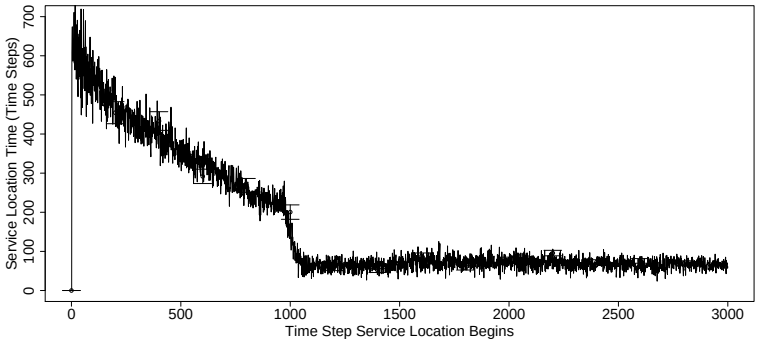


FIGURE B.3: Service location time, frequent service connection reorganisation

B.3 Service Location Time When Preserving Topology

B.3.1 Pipeline

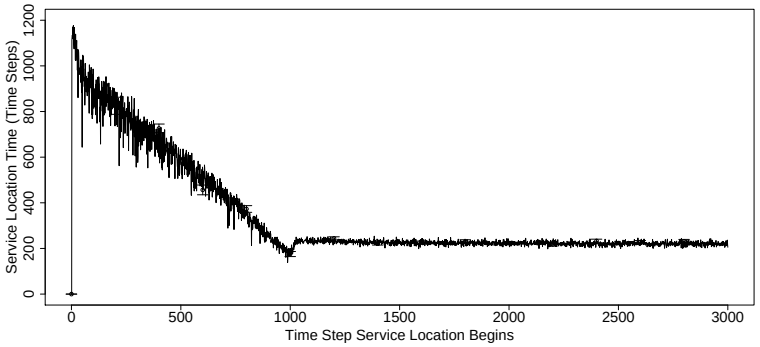


FIGURE B.4: Service location time, random reorganisation, pipeline

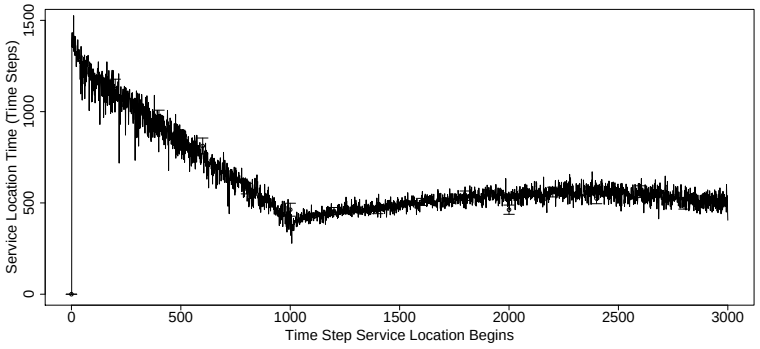


FIGURE B.5: Service location time, full service connection reorganisation, hierarchy

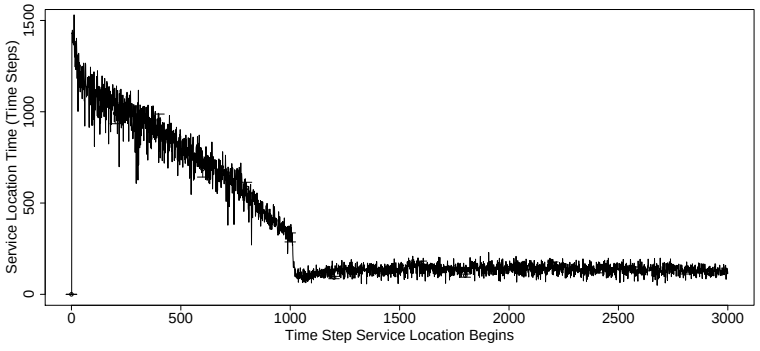


FIGURE B.6: Service location time, frequent service connection reorganisation, pipeline

B.3.2 Hierarchy

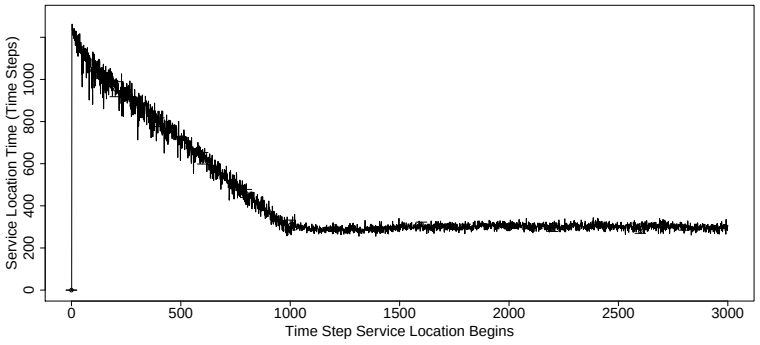


FIGURE B.7: Service location time, random reorganisation, hierarchy

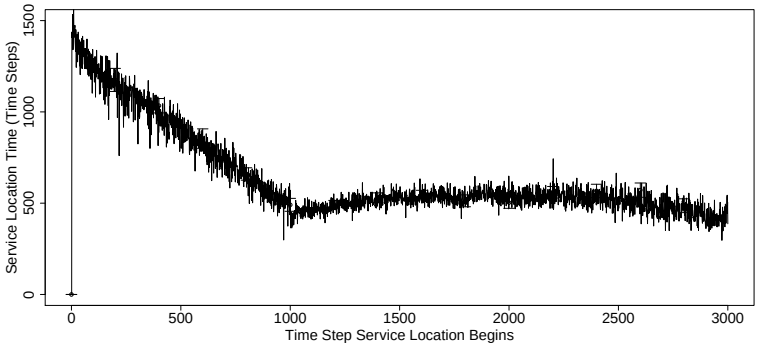


FIGURE B.8: Service location time, full service connection reorganisation, pipeline

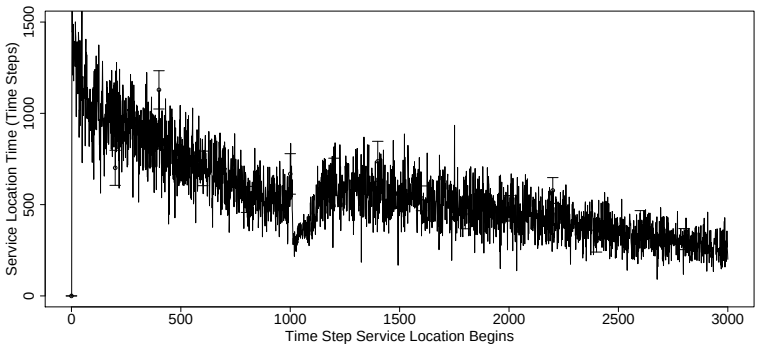


FIGURE B.9: Service location time, frequent service connection reorganisation, hierarchy

References

[1] S. Abdallah and V. Lesser. Learning the task allocation game. In *Proceedings of the Fifth International Joint Conference on Autonomous Agents and MultiAgent Systems*, pages 850–857, 2006.

[2] S. Abdallah and V. Lesser. Multiagent reinforcement learning and self-organization in a network of agents. In *Proceedings of the Sixth International Joint Conference on Autonomous Agents and MultiAgent Systems*, pages 172–179, 2007.

[3] D. Allsopp, J. Bradshaw, P. Beautement, E. Durfee, N. Suri, M. Kirton, C. Knoblock, A. Tate, and C. Thompson. Coalition Agents Experiment: Multi-Agent Co-operation in an International Coalition Setting. *IEEE Intelligent Systems, Special Issue on Knowledge Systems for Coalition Operations (KSCO)*, 17(3):26–35, 2002.

[4] G. Altekar, S. Dwarkadas, J. Huelsenbeck, and F. Ronquist. Parallel metropolis coupled Markov chain Monte Carlo for Bayesian phylogenetic inference. *Bioinformatics*, 20(3):407–415, 2004.

[5] E. Argente, V. Julian, and V. Botti. Multi-Agent System Development Based on Organizations. *Electronic Notes in Theoretical Computer Science, Proceedings of the First International Workshop on Coordination and Organisation*, 150(3):55–71, 2006.

[6] A. Barabási and R. Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, 1999.

[7] G. Beavers and H. Hexmoor. Teams of Agents. In *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics*, pages 574–582, 2001.

[8] D. Ben-Ami and O. Shehory. Evaluation of distributed and centralized agent location mechanisms. In M. Klusch, S. Ossowski, and O. Shehory, editors, *Cooperative Information Agents VI, Proceedings of the Sixth International Workshop on Cooperative Information Agents*, volume 2446 of *Lecture Notes in Computer Science*, pages 264–278. Springer, 2002.

[9] C. Bernon, V. Chevrier, V. Hilaire, and P. Marrow. Applications of Self-Organising Multi-Agent Systems: An Initial Framework for Comparison. *Informatica*, 30(1):73–82, 2006.

[10] C. Bernon, M. Gleizes, and G. Picard. Enhancing Self-Organising Emergent Systems Design with Simulation. In G. O'Hare, A. Ricci, M. O'Grady, and O. Dikenelli, editors, *Agent World VII, Proceedings of the Seventh International Conference in Engineering Societies in the Agent World*, volume 4457 of *Lecture Notes In Computer Science*, pages 284–299. Springer, 2007.

[11] A. Berson. *Client/Server Architecture*. McGraw-Hill, 2nd edition, 1996.

[12] K. Binder and D. Heermann. *Monte Carlo simulation in statistical physics: an introduction*. Springer, 2010.

[13] B. Blankenburg, R. Dash, S. Ramchurn, N. Jennings, and M. Klusch. Trusted Kernel-Based Coalition Formation. In *Proceedings of the Fourth International Joint Conference on Autonomous Agents and MultiAgent Systems*, pages 989–996, 2005.

[14] A. Bond and L. Gasser. An analysis of problems and research in DAI. In A. Bond and L. Gasser, editors, *Readings in Distributed Artificial Intelligence*, pages 3–35. Morgan Kaufmann Publishers, 1988.

[15] J. Boyan and M. Littman. Package routing in dynamically changing networks: a reinforcement learning approach. *Advanced Neural Information Processing Systems*, 6:671–678, 1994.

[16] J. Bradshaw, G. Klein, R. Hoffman, P. Feltovich, and D. Woods. Ten challenges for making automation a "team player" in joint human-agent activity. *IEEE Intelligent Systems*, 19(6):91–95, 2004.

[17] C. Brooks and E. Durfee. Congregation Formation in information Economies. In *AAAI Workshop on Artificial Intelligence in Electronic Commerce*, pages 62–85, 1999.

[18] C. Brooks and E. Durfee. Congregating and Market Formation. In *Proceedings of the First International Joint Conference on Autonomous Agents and MultiAgent Systems*, pages 96–103, 2002.

[19] C. Brooks and E. Durfee. Congregation Formation in Multiagent Systems. *Autonomous Agents and Multiagent Systems*, 7(1-2):145–170, 2003.

[20] C. Brooks, E. Durfee, and A. Armstrong. An Introduction to Congregating in Multiagent Systems. In *Proceedings of the Fourth International Conference on Multiagent Systems*, pages 79–86, 2000.

[21] L. Buşonui, R. Babuška, and Schutter. Multi-agent Reinforcement Learning: An Overview. In D. Srinivasan and L. Jain, editors, *Innovations in Multi-Agent Systems and Applications*, volume 310 of *Studies in Computational Intelligence*, pages 183–221. Springer, 2010.

[22] Scott Camazine, Jean-Louis Deneubourg, Nigel R Franks, James Sneyd, Guy Theraulaz, and Eric Bonabeau. *Self-Organization in Biological Systems*. Princeton Studies in Complexity. Princeton University Press, 2001.

[23] K. Carley and L. Gasser. Computational Organization Theory. In *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*, pages 299–330. The MIT Press, 1999.

[24] G. Chalkiadakis, E. Elkind, M. Polukarov, and N. Jennings. The Price of Democracy in Coalition Formation. In *Proceedings of the Eighth International Conference on Autonomous Agents and MultiAgent Systems*, pages 401–408, 2009.

[25] B. Chandrasekaran. Natural and Social System Metaphors for Distributed Problem Solving: Introduction to the Issue. *IEEE Transactions on Systems, Man and Cybernetics*, 11(1):1–5, 1981.

[26] P. Cohen, H. Levesque, and I. Smith. On team formation. In J. Hintikka and R. Tuomela, editors, *Contemporary Action Theory: Social Action*, volume 2 of *Synthese Library*, pages 87–114. Springer, 1997.

[27] D. Corkill and S. Lander. Diversity in Agent Organizations. *Object Magazine*, 8(4):41–47, 1998.

[28] V. Dang. *Coalition Formation and Operation in Virtual Organisations*. Ph.D. Thesis, University of Southampton, 2004.

[29] T. De Wolf and T. Holvoet. Emergence and Self-Organisation: a statement of similarities and differences. In *Proceedings of the International Workshop on Engineering Self-Organising Applications*, pages 96–110, 2004.

[30] T. De Wolf and T. Holvoet. Emergence versus Self-Organisation: Different Concepts but Promising When Combined. In S. Brueckner, G. Serugendo, A. Karageorgos, and R. Nagpal, editors, *Engineering Self Organising Systems I, Methodologies and Applications*, volume 3464 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2005.

[31] K. Decker. TAEMS: A Framework for Environment Centered Analysis & Design of Coordination Mechanisms. In *Foundations of Distributed Artificial Intelligence*, pages 429–448. John Wiley & Sons, 1996.

[32] C. Dellarocas and M. Klein. Civil Agent Societies: Tools for Inventing Open Agent-Mediated Electronic Marketplaces. In A. Moukas, F. Ygge, and C. Sierra, editors, *Agent Mediated Electronic Commerce II, Towards Next-Generation Agent-Based Electronic Commerce Systems*, volume 1788 of *Lecture Notes In Computer Science*, pages 24–39. Springer, 2000.

[33] D. Didona, D. Carnevale, P. Romano, and S. Galeani. An Extremum Seeking Algorithm for Message Batching in Total Order Protocols. In *Proceedings of The Sixth International Conference on Self-Adaptive and Self-Organizing Systems*, pages 89–98, 2012.

[34] J. Diggelen, J. Bradshaw, M. Johnson, A. Uszok, and P. Feltovich. Implementing Collective Obligations in Human-Agent Teams using KAoS Policies. In J. Padget, A. Artikis, W. Vasconcelos, K. Stathis, V. Torres da Silva, E. Matson, and Axel Polleres, editors, *Coordination, Organizations, Institutions and Norms in Agent Systems V, Proceedings of the Eleventh International Workshop on Coordination, Organizations, Institutions, and Norms, Co-located with AAMAS 2009*, volume 6069 of *Lecture Notes In Computer Science*, pages 36–52. Springer, 2009.

[35] P. Erdős and A. Rényi. On The Evolution of Random Graphs. In *Publication of The Mathemetical Institute of The Hungarian Academy of Sciences*, pages 17–61, 1960.

[36] European Telecommunications Standards Institute. *Digital cellular telecommunications system (Phase 2+); Universal Mobile Telecommunications System*

(UMTS); LTE; Network architecture. European Telecommunications Standards Institute, 11.5.0 edition, 2013.

[37] K. Fischer. Agent-based Design of Holonic Manufacturing Systems. *Journal of Robotics and Autonomous Systems*, 27(1-2):3–13, 1999.

[38] D. Fitoussi and M. Tennenholtz. Choosing Social Laws for Multi-Agent Systems: Minimality and Simplicity. *Artificial Intelligence*, 199:61–101, 2000.

[39] M. Fox. Organization structuring: Designing Large Complex Software. Technical Report CMU-CS-79-155, Carnegie-Mellon University, Department of Computer Science, 1979.

[40] M. Fox. An Organizational View of Distributed Systems. *IEEE Transactions on Systems, Man and Cybernetics*, 11(1):70–80, 1981.

[41] M. Gardner. The fantastic combinations of John Conway's new solitaire game "life". *Scientific American*, 223(4):120–123, 1970.

[42] M. Gaston and M. desJardins. Agent-Organized Networks for Dynamic Team Formation. In *Proceedings of the Fourth International Joint Conference on Autonomous Agents and MultiAgent Systems*, pages 230–237, 2005.

[43] C. Gershenson, C. Apostel, and V. Brussel. Towards Self-organizing Bureaucracies. *International Journal of Public Information Systems*, 4(1):1–24, 2008.

[44] A. Getling. *Rayleigh-Bénard Convection: Structures and Dynamics*. World Scientific Publishing, 1998.

[45] P. Glasserman. *Monte Carlo methods in financial engineering*, volume 53 of *Stochastic Modelling and Applied Probability*. Springer, 2003.

[46] N. Griffiths. Supporting Cooperation Through Clans. In *Proceedings of the Second IEEE UK&RI Chapter Conference on Cybernetic Intelligence — Challenges and Advances for Systems, Man and Cybernetics*, 2003.

[47] N. Griffiths. Cooperative Clans. *Kybernetes*, 34(9-10):1384–1403, 2005.

[48] N. Griffiths and M. Luck. Coalition Formation through Motivation and Trust. In *Proceedings of the Second International Joint Conference On Autonomous Agents and MultiAgent Systems*, pages 17–24, July 2003.

[49] B. Grosz and S. Kraus. Collaborative Plans for Complex Group Action. *Artificial Intelligence*, 86(2):269–357, 1996.

[50] H. Guo, F. Tao, L. Zhang, S. Su, and N. Si. Correlation-aware web services composition and QoS computation model in virtual enterprise. *International Journal of Advanced Manufacturing Technology*, 51(5-8):817–827, 2010.

[51] M. Hannoun, O. Boissier, J. Sichman, and C. Sayettat. MOISE: An Organizational Model for Multi-Agent Systems. In M. Monard and J. Sichman, editors, *Advances in Artificial Intelligence, Proceedings of the Seventh International Ibero-American Conference on AI*, volume 1952 of *Lecture Notes in Computer Science*, pages 156–165. Springer, 2000.

[52] T. Hey and A. Trefethen. The UK e-Science Core Programme and the Grid. *Future Generation Computer Systems*, 18(8):1017–1031, October 2002.

[53] T. Hey and A. Trefethen. Cyberinfrastructure for e-Science. *Science*, 308(5723):817–821, 2005.

[54] J. Holland. *Emergence: From Chaos To Order*. Addison Wesley Longman Inc., 1998.

[55] R. Holzer, H. De Meer, and C. Bettstetter. On Autonomy and Emergence in Self-Organizing Systems. In K Hummel and J Sterbenz, editors, *Self-Organizing Systems, Proceedings of the Third International Workshop on Self-Organizing Systems*, volume 5343 of *Lecture Notes in Computer Science*, pages 157–169. Springer, 2008.

[56] B. Horling and V. Lesser. A survey of multi-agent organizational paradigms. *The Knowledge Engineering Review*, 19(4):281–316, 2005.

[57] B. Horling, R. Mailler, J. Shen, R. Vincent, and V. Lesser. Using Autonomy, Organizational Design and Negotiation in a Distributed Sensor Network. In *Distributed Sensor Networks: A multiagent perspective*, pages 139–183. Springer, 2003.

[58] B. Horling, R. Vincent, R. Mailler, J. Shen, R. Becker, K. Rawlins, and V. Lesser. Distributed Sensor Network For Real Time Tracking. In *Proceedings of the Fifth International Conference on Autonomous Agents*, pages 417–424. ACM Press, 2001.

[59] T. Ishida, L. Gasser, and M. Yokoo. Organizational Self-Design of Distributed Production Systems. *IEEE Transactions on Knowledge and Data Engineering*, 4(2):123–134, 1992.

[60] S. Johansson. Valuation-based coalition formation in multi-agent systems. In J. Liu, N. Zhong, Y. Tang, and P. Wang, editors, *Agent Engineering*, volume 43 of *Series in Machine Perception and Artificial Intelligence*, pages 149–173. World Scientific, 2001.

[61] J. Kephart and D. Chess. The Vision of Autonomic Computing. *Computer*, 36(1):41–50, 2003.

[62] J. Kinnebrew and G. Biswas. Efficient Allocation of Hierarchically-Decomposable Tasks in a Sensor Web Contract Net. In *Proceedings of the 2009 IEEE/WIC/ACM International Joint Conference on Web Intelligence and Intelligent Agent Technology*, pages 225–232, 2009.

[63] J. Kittock. Emergent Conventions and the Structure of Multi-Agent Systems. In *Proceedings of the 1993 Complex Systems Summer School, Santa Fe Institute Studies in the Sciences of Complexity*, pages 507–521, 1993.

[64] J. Ko, C. Lu, M. Srivastava, J. Stankovic, A. Terzis, and M. Welsh. Wireless sensor networks for healthcare. *Proceedings of the IEEE*, 98(11):1947–1960, 2010.

[65] A. Koestler. *The Ghost In The Machine*. Hutchinson Publishing Group, 1967.

[66] R. Kota. *Self-Adapting Agent Organisations*. Ph.D. Thesis, University of Southampton, 2009.

[67] R. Kota, N. Gibbins, and N. Jennings. A Generic Agent Organisation Framework For Autonomic Systems. In A. Vasilakos, R. Beraldi, R. Friedman, and M. Mamei, editors, *Autonomic Computing and Communications Systems, Proceedings of the Third International Conference on Software Testing*, volume 23 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 203–219. Springer, 2009.

[68] R. Kota, N. Gibbins, and N. Jennings. Decentralised Structural Adaptation in Agent Organisations. In George Vouros, A. Artikis, K. Stathis, and J. Pitt, editors, *Organized Adaption in Multi-Agent Systems, Proceedings of the First International Workshop on Organized Adaptation in Multi-Agent Systems 2008*, volume 5368 of *Lecture Notes In Computer Science*, pages 54–71. Springer, 2009.

[69] R. Kota, N. Gibbins, and N. Jennings. Self-Organising Agent Organisations. In *Proceedings of the Eighth International Conference on Autonomous Agents and Multiagent Systems*, pages 797–804, 2009.

[70] R. Kota, N. Gibbins, and N. Jennings. Decentralised Approaches for Self-Adaptation in Agent Organisations. *ACM Transactions on Autonomous and Adaptive Systems*, 7(1):1–28, 2012.

[71] D. Krackhardt and K. Carley. A PCANS Model of Structure in Organization. In *Proceedings of the International Symposium on Command and Control Research and Technology*, pages 113–119, 1998.

[72] S. Kraus, O. Shehory, and G. Taase. Coalition Formation with Uncertain Heterogenous Information. In *Proceedings of the Second International Joint Conference on Autonomous Agents and MultiAgent Systems*, pages 1–8, July 2003.

[73] A. Kumar, J. Xu, and A. Zegura. Efficient and scalable query routing for unstructured peer-to-peer networks. In *Proceedings of the 24th Annual Joint Conference of the IEEE Computer and Communications Societies*, pages 1162–1173, 2005.

[74] V. Lesser. Reflections on the Nature of Multi-Agent Coordination and Its implications for an Agent Architecture. *Autonomous Agents and Multiagent Systems*, 1(1):89–111, 1998.

[75] M. Luck, P. McBurney, O. Shehory, and S. Willmott. *Agent Technology: Computing as Interaction, A Roadmap for Agent Based Computing.* AgentLink III, 2005.

[76] S. Mahmoud. *Metanorms, Topologies, and Adaptive Punishment in Norm Emergence.* Ph.D. Thesis, King's College London, 2013.

[77] S Mahmoud, J. Keppens, M. Luck, and N. Griffiths. Norm Establishment via Metanorms in Network Topologies. In *Proceedings of the International Conference on Web Intelligence and Intelligent Agent Technology*, pages 25–28, 2011.

[78] M. Mamei, M. Vasirani, and F. Zambonelli. Self-Organizing Spatial Shapes in Mobile Particles: The TOTA Approach. In S. Brueckner, G. Serugendo, A. Karageorgos, and R. Nagpal, editors, *Engineering Self-Organising Systems, Methodologies and Applications*, volume 3464 of *Lecture Notes In Computer Science*, pages 138–153. Springer, 2005.

[79] P. Mathieu, J. Routier, and Y. Secq. Principles for Dynamic Multi-Agent Organizations. In K. Kuwabara and J. Lee, editors, *Intelligent Agents and Multi-Agent Systems, Proceedings of the Fifth Pacific Rim International Workshop on*

Multi-Agents, volume 2413 of *Lecture Notes in Computer Science*, pages 109–122. Springer, 2002.

[80] F. Maturana, W. Shen, and D. Norrie. Metamorph: An adaptive agent-based architecture for intelligent manufacturing. *International Journal of Production Research*, 37(10):2159–2174, 1999.

[81] J. Miller and S. Page. *Complex Adaptive Systems*. Princeton University Press, 2007.

[82] J. Moffett. Control Principles and Role Hierarchies. In *Proceedings of the Third ACM Workshop on Role-Based Access Control*, pages 63–69. ACM Press, 1998.

[83] C. Mooney. *Monte Carlo Simulation*, volume 116 of *Quantitative Applications in the Social Sciences*. SAGE Publications, 1997.

[84] Y. Moses and M. Tennenholtz. Artificial Social Systems. *Computers and AI*, 14(6):533–562, 1995.

[85] E. Newman. The Structure and Function of Complex Networks. *Society of Industrial and Applied Mathematics Review*, 45(2):167–256, 2003.

[86] A Nowé, P. Vrancx, and Y. Hauwere. Game Theory and Multi-agent Reinforcement Learning. In M. Wiering and M. Otterlo, editors, *Reinforcement Learning*, volume 12 of *Adaptation, Learning, and Optimization*, pages 441–470. Springer, 2012.

[87] J. Odell. Agent and Complex Systems. *Journal of Object Technology*, 1(2):35–45, 2002.

[88] L. Panait and S. Luke. A Pheromone-Based Utility Model for Collaborative Foraging. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multi-Agent Systems*, pages 36–43, 2004.

[89] L. Panait and S. Luke. Cooperative Multi-Agent Learning: The State of the Art. *Autonomous Agents and Multiagent Systems*, 11(3):387–434, 2005.

[90] L. Parker. Designing Control Laws for Cooperative Agent Teams. In *Proceedings of IEEE International Conference on Robotics and Automation*, pages 582–587, 1993.

[91] H. Parunak and S. Brueckner. Entropy and Self-Organization in Multi-Agent Systems. In *Proceedings of the International Conference on Autonomous Agents*, pages 124–130, 2001.

[92] M. Pechoucek, V. Marík, and J. Bárta. CPlanT: An Acquaintance Model-Based Coalition Formation Multi-agent System. In B. Dunin-Keplicz and E. Nawarecki, editors, *From Theory to Practice in Multi-Agent Systems, Proceedings of the Second International Workshop of Central and Eastern Europe on Multi-Agent Systems*, volume 2296 of *Lecture Notes in Computer Science*, pages 234–241. Springer, 2001.

[93] L. Peshkin and V. Savova. Reinforcement Learning for Adaptive Routing. In *Proceedings of the International Joint Conference on Neural Networks*, pages 1825–1830, 2002.

[94] T. Rahwan. *Algorithms for Coalition Formation in Multi-Agent Systems*. Ph.D. Thesis, University of Southampton, 2007.

[95] T. Rahwan, T. Michalak, M. Wooldridge, and N. Jennings. Anytime Coalition Structure Generation in Multi-Agent Systems with Positive or Negative Externalities. *Artificial Intelligence*, 186:95–122, 2012.

[96] S. Ramchurn, D Huynh, and N. Jennings. Trust in Multi-Agent Systems. *The Knowledge Engineering Review*, 19(1):1–25, 2004.

[97] S. Ramchurn, N. Jennings, T. Rahwan, and V. Dang. Near-optimal anytime coalition structure generation. In *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence*, pages 2365–2371, 2007.

[98] J. Rosenschein and G. Zlotkin. *Rules of Encounter: Designing Conventions for Automated Negotiation Among Computers*. The MIT Press, 1994.

[99] S. Russell and P. Norvig. *Artificial Intelligence - A Modern Approach*. Pearson, 3rd edition, 2010.

[100] T. Sandholm, K. Larson, M. Andersson, O. Shehory, and F. Tohme. Coalition Structure Generation with Worst Case Guarantees. *Artificial Intelligence*, 111(1-2):209–238, 1999.

[101] T. Sandholm and V. Lesser. Coalitions Amoung Computationally Bounded Agents. *Artificial Intelligence, Special Issue on Economic Principles of Multi-Agent Systems*, 94(1–2):99–137, 1997.

[102] N. Schurr, J. Marecki, J. Lewis, M. Tambe, and P. Scerri. The DEFACTO system: Training Tool for Incident Commanders. In *Proceedings of the Seventeenth Conference on Innovative Applications of Artificial Intelligence*, pages 1555–1562, 2005.

[103] N. Schurr, P. Patil, F. Pighin, and M. Tambe. Using Multiagent Teams to Improve the Training of Incident Commanders. In *Proceedings of the Fifth International Joint Conference on Autonomous Agents and MultiAgent Systems*, pages 1490–1497, 2006.

[104] G. Serugendo, M. Gleizes, and A. Karageorgos. Self-Organisation In Multi-Agent Systems. *The Knowledge Engineering Review*, 20(2):165–189, 2005.

[105] G. Serugendo, M. Gleizes, and A. Karageorgos. Self-organisation and emergence in MAS: An overview. *Informatica*, 30(1):45–54, 2006.

[106] O. Shehory and S. Kraus. Coalition Formation Among Autonomous Agents: Strategies and Complexity. In C. Castelfranchi and J.-P. Müller, editors, *From Reaction to Cognition, Proceedings of the Fifth European Workshop on Modelling Autonomous Agents in a Multi-Agent World*, volume 957 of *Lecture Notes in Computer Science*, pages 55–72. Springer, 1995.

[107] O. Shehory and S. Kraus. Task allocation via coalition formation among autonomous agents. In *Proceedings of the Twenty First International Joint Conference on Artificial Intelligence*, pages 655–661, 1995.

[108] O. Shehory and S. Kraus. Methods for Task Allocation via Agent Coalition Formation. *Artificial Intelligence*, 101(1-2):165–200, 1998.

[109] Y. Shoham and M. Tennenholtz. On Social Laws for Artificial Agent Societies: Off-Line Design. *Artificial Intelligence, Special Volume on Computational Research on Interaction and Agency*, 73(1-2):231–252, 1995.

[110] G. Simon, M. Maróti, Á. Lédeczi, G. Balogh, B. Kusy, A. Nádas, G. Pap, J. Sallai, and K. Frampton. Sensor network-based countersniper system. In *Proceedings of the Second International Conference on Embedded Networked Sensor Systems*, pages 1–12. ACM, 2004.

[111] M. Sims, C. Goldman, and V. Lesser. Self-Organization through Bottom-up Coalition Formation. In *Proceedings of the Second International Joint Conference on Autonomous Agents and MultiAgent Systems*, pages 867–874, 2003.

[112] A. Smith. *An Inquiry into the Nature and Causes of the Wealth of Nations*. Liberty Fund, 1981.

[113] A. Smith. Applications of the Self-Organising Map to Reinforcement Learning. *Neural Networks*, 15:1107–1124, 2002.

[114] R. Smith. The Contract Net Protocol: High-Level Communication and Control in a Distributed Problem Solver. *IEEE Transactions on Computers*, C-29(12):1104–1113, December 1980.

[115] L. Soh and X. Li. A learning-based coalition formation model for multiagent systems. In *Proceedings of the Second International Joint Conference on Autonomous Agents and MultiAgent Systems*, pages 1120–1121, Melbourne, Australia, 2003.

[116] M. Tambe. Towards Flexible Teamwork. *Journal of Artificial Intelligence Research*, 7:83–124, 1997.

[117] J. Thompson. *Organizations in Action*. McGraw-Hill, 1967.

[118] H. Tianfield and R. Unland. Towards Self-Organization In Multi-Agent Systems and Grid Computing. *Multiagent and Grid Systems*, 1(2):89–95, 2005.

[119] G. Valetto, P. Snyder, D. Dubois, and E. Nitto. A Self-Organized Load-Balancing Algorithm Overlay-Based Decentralised Service Networks. In *Proceedings of the Fifth International Conference on Self-Adaptive and Self-Organizing Systems*, pages 168–177, 2011.

[120] T. Voice, M. Polukarov, and N. Jennings. Coalition Structure Generation Over Graphs. *Journal of Artificial Intelligence Research*, 45:165–196, 2012.

[121] D. Watts and S. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393:440–442, 1998.

[122] M. Weerdt, Y. Zhang, and T. Klos. Multiagent Task Allocation in Social Networks. *Autonomous Agents and Multiagent Systems*, 25(1):48–86, 2012.

[123] S. Willmott, J. Dale, B. Burg, P. Charlton, and P. O'Brien. Agentcities: A Worldwide Open Agent Network. *Agentlink Newsletter*, 8:13–15, 2001.

[124] M. Wooldridge. *An Introduction to MultiAgent Systems*. John Wiley & Sons, second edition, 2009.

[125] J. Yick, B. Mukherjee, and D. Ghosal. Wireless sensor network survey. *The International Journal of Computer and Telecommunications Networking*, 52(12):2292–2330, 2008.

[126] F. Zambonelli, N. Jennings, and M. Wooldridge. Organisational abstractions for the analysis and design of multi-agent systems. In P. Ciancarini and M. Wooldridge, editors, *Proceedings of The First International Workshop on Agent-Oriented Software Engineering 2000, Organisational Abstractions for the Analysis and Design of Multi-agent Systems*, volume 1957 of *Lecture Notes in Computer Science*, pages 407–422. Springer, 2001.

[127] F. Zambonelli, N. Jennings, and M. Wooldridge. Developing multiagent systems: the Gaia Methodology. *ACM Transactions on Software Engineering and Methodology*, 12(3):317–370, 2003.

[128] C. Zhang, S. Abdallah, and V. Lesser. Integrating Organizational Control into Multi-Agent Learning. In *Proceedings of the Eighth International Conference on Autonomous Agents and MultiAgent Systems*, pages 757–764, 2009.

[129] T. Ziermann, N. Mühleis, S. Wildermann, and J. Teich. A self-organizing distributed reinforcement learning algorithm to achieve fair bandwidth allocation for priority-based bus communication. In *Proceedings of the Thirteenth IEEE International Symposium on Object/Component/Service-Oriented Reat-Time Distributed Computing Workshops*, pages 11–20, 2010.